Evaluation of Modified Condition/Decision Coverage in Testing a Web Server

Nan LI (11139552)

Research Thesis 2014

M.Sc.in Computer Science (Software Engineering)

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

County Co.Kildare, Ireland

A thesis submitted in partial fulfillment
of the requirements for the
M.Sc.in Computer Science (Software Engineering)

Supervisor: Dr. Stephen Brown

January 2014

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Software Engineering, is entirely my own work and has not been taken from the work of the others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Nan Li Date: 28/1/2014

Acknowledgement

I would like to thank Dr. Stephen Brown and Dr. John G.Keating at National University of Ireland, Maynooth for their support and input of the project.

Abstract

Modified Condition/Decision Coverage (MC/DC) is a structural coverage criterion widely used in testing aviation software. Aviation software has a high level of state-based behavior, typically implemented with complex Boolean expressions. MC/DC was developed to provide the benefits of exhaustive testing of the Boolean expressions, without the overhead. Web servers are also state-based systems, and the purpose of this research is to determine whether the benefits of MC/DC also apply to these systems. In this paper, a unit-testing case-study on a typical web server Culture Object Management Web Server (COMWS) was performed to evaluate the effectiveness of MC/DC in this context. For each method in COMWS, Black-Box testing was carried out first, followed by MC/DC testing. The Black-Box testing was used as a control group for MC/DC testing. The comparison focuses on three criteria: Testing Cost, Faults Found and Program Coverage. Based on the experimental results and comparative evaluation, two main conclusions were obtained: MC/DC is an additional effective testing technique to complement Black-Box testing for testing web servers, and it provides extra test capability where some of the variables used in Boolean expression are not directly derived from the input parameters.

Table of Contents

1. Introduction

1.1 Motivation

In recent years, the number of web applications has grown extraordinarily on a worldwide basis. As web applications become more complex, their quality and reliability become crucial, especially for back-end web servers. Web servers are software programs that operate essentially independently from the clients (e.g. from the specific browser) to offer services over the Internet [LiH00]. If a web server has low quality, the web server may provide incorrect information to a client user or cause data loss in the back-end. However, relatively little attention has been paid to research on web server testing.

Different testing approaches can be used to test web servers, such as unit testing, integration testing, and system testing. This project focuses on unit testing, used to check the correctness of methods or classes in the web server.

When performing unit testing on a web server, two types of measurement can be considered: requirements coverage measurement (Black-Box testing) and structural coverage measurement (White-Box testing). Requirements coverage measurement aims to check whether the tested web server method meets its specification (which describes its behavior and features) or not, while structural coverage measurement provides a means to confirm that "the requirements-based test procedures exercised the code structure of the tested method". Requirements coverage analysis should be accomplished and reviewed before structural coverage analysis begins [HVC+01].

Modified Condition / Decision Coverage (MC/DC) is a structural coverage criterion requiring the basic requirement for Decision/Condition Coverage (DCC), and also needing that each condition within a decision is shown by execution to independently and correctly affect the outcome of the decision [ChM94].

This criterion is widely used in aviation software testing. To be certified by the FAA (Federal Aviation Administration), aviation software must satisfy standard DO-178B [HaV01]. Software development processes are specified in this standard for software of varying levels of criticality. With respect to testing, the most critical (Level A) software, which is defined as that which could prevent continued safe flight and landing of the aircraft, must satisfy MC/DC.

The importance of MC/DC in aviation software testing field arises from the nature of current flight control programs where the actuator commands depend on the **state** of the system [Whi01]. The state of system is controlled by up to several hundred very long Boolean expressions. Testing such software by using multiple-condition

coverage is infeasible, which is due to the large number of tests. MC/DC reduces the number of test cases while providing similar test coverage. It was developed to provide many of the benefits of exhaustive testing of Boolean expressions without requiring the cost of exhaustive testing [ChM94, HaV01].

Web servers are also state-based systems. They indentify the next state of a web application based on the current state and the user input. So based on the role MC/DC plays in testing aviation software (as a stated-based system), MC/DC was selected as a good candidate for testing web servers in this project.

The motivation of this project is to evaluate MC/DC by assessing its effectiveness in testing a typical web server. The effectiveness is defined by measuring the additional cost and the additional benefits of MC/DC compared to Black-Box testing.

1.2 Software under Test

An Foras Feasa proposed a project called Cultural Objects Management Project (COMP) which aims to help users to manage their cultural objects and reduce the difficulty in contributing cultural data to professional data repositories.

In order to better maintain and research culture objects, national museums cooperate with these large repositories to store and manage different metadata (such as Europeana [DGH+10], DC [Dub03]) to record these objects' information. But for private providers, such as small museum curators and private collectors, the procedure of contributing cultural data is more difficult and they have to learn how to exchange the data and understand different data structures to represent their cultural objects. So it is necessary to create a bridge between these private providers and large data repositories. COMP was designed to provide such service and give convenience to those private providers. It is intended that COMWS will be used by various national bodies, so it is important that it operates correctly.

Cultural Object Management Web Server (COMWS) is the core of COMP and it provides three services:

- Acts as the back-end server to communicate with the front-end cultural objects management website.
- Includes servlet, which is used to communicate with mobile application in order to help users to manage their cultural objects by using their mobile phone.
- Acts as the intermediate software to connect with these large repositories to do the cultural objects data exchange.

1.3 Method

In order to evaluate whether MC/DC is actually effective in testing web server, a related experiment - testing COMWS, was implemented to verify the proposed

viewpoint. Testing such web server includes 2 parts:

- Black-Box Testing: testing whether the code of COMWS meets the specification or not (focus on the specification)
  Result data collected:
  1. BB tests number
  2. The work (cost time) to write these BB tests
  3. Faults found number
  4. Branch coverage of tested program
  5. Code coverage of tested program

- MC/DC Testing: using MC/DC coverage criterion to test the implementation of COMWS (focus on the inner structure of the code)
  Result data collected:
  1. MC/DC tests number
  2. The work (cost time) to write these MC/DC tests
  3. Faults found number
  4. Branch coverage of tested program
  5. Code coverage of tested program
  6. Extra MC/DC tests number to find faults which BB testing didn't find

Result data of Black-Box Testing was compared with that of MC/DC Testing to evaluate the effectiveness of MC/DC in testing COMWS. The evaluation focused on three criteria:

- Testing Cost: aims to find whether MC/DC Testing required more works than Black Box Testing
- Faults Found: aims to find whether MC/DC Testing found extra faults or not
- Coverage Area: aims to find the influence MC/DC Testing had on the branch coverage and code coverage

By doing such evaluation, the comparison result can provide the evidence to verify whether MC/DC is actually effective in testing web server or not.

1.4 Report Overview

The rest of this report is organized as follows:

- Section 2 provides background into the problems of evaluating the effectiveness of MC/DC for testing the COMWS. It gives the explanation of MC/DC and proposes the testing solution to these problems.
- Section 3 describes the software and tools used in the testing solution.
- Section 4 gives an overview of COMWS (functionality and a description of classes).
- Section 5 details the testing design for COMWS and the implementation of the testing solution.

- Section 6 describes and draws conclusions from the testing results.
- Section 7 gives the evaluation based on these testing results.
- Section 8 provides a critical analysis, presents conclusions, and makes suggestions for future work.

2. Background & Proposal

This section describes the background details, explaining why MC/DC was selected as a good candidate test technique for testing a web server. It describes the fundamental software testing concepts used, and provides a detailed explanation of MC/DC. It analyzes the reasons for using MC/DC to test a web server (COMWS) and also describes previous work on MC/DC testing. Finally, it describes the proposal of my project.

2.1 Related Software Testing Concepts

Some essential related Software Testing Concepts are shown as below:

- **Software Quality**

  With the advancement and development of new software product, a major process, which needs to be systematic scrutinize is the quality aspects. In the world of competitive business market, a company has to make sure that the user or clients are getting competitive and effective software products all the time. Customers are not going to buy any low quality software products, and the only way to promise that the software products are ready for the market is to make sure that they pass the professional quality evaluation process.

  Software Quality refers to:
  The degree of conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software. [Gal04]

  This definition indicates that high quality software needs to meet the following three requirements:
  - Specific functional requirements, which refer mainly to the outputs of the software system.
  - The software quality standards mentioned in the contract.
  - Good Software Engineering Practices (GSEP) [Gal04], reflecting state-of-the-art professional practices that are not explicitly mentioned in the contract.

  In order to produce high quality software (meets the three requirements above),

a series of professional activities are proposed. Software testing is one of these activities and has a close relationship with software quality. Software testing is the first step applied to control the software product's quality before its shipment or installation at the customer's premises. Suitable test procedures can bring the tested software to an acceptable level of quality after correction of the identified errors and retesting [BTL^{+}11], by contraries, unreasonable test procedures may leads to poor software quality, which will cause more failures, increase development costs and delay in attaining product stability [BTL^{+}11]. So software testing provides a great impact on software quality and developing suitable software test procedures is really necessary.

- **Testing Techniques**

  - **Black-Box (Functional) Testing and White-Box (Structural) Testing:** Black-Box Testing is based on the program specification only (Software function) and doesn't consider the inner code of the tested program. All the test cases are generated from specification. The goal of Block-Box Testing is to verify that the program meets the specified requirements.

    White-Box Testing is based on the inner program code (structure) & the specification and all the test cases are designed to exercise the implementation. The goal of White-Box Testing is to ensure that executing the components (software can be viewed as a set of components) always results in the correct output value. White-Box Testing is used to enhance Black-Box testing and try to improve coverage of the internal components that form the program.

  - **Decisions and Conditions:** a decision is a compound Boolean expression that controls the flow of the program (always exists in the '*if*', '*for*' and '*while*' statements). A condition is a leaf-level Boolean expression (cannot be broken down into a simpler Boolean expression) that can be used to make up a decision. A decision consists of one or more conditions.

  - **Decision/Condition Coverage (DCC):** The test cases for this white-box testing technique ensure that every decision in the program is true and false at least once, and that every condition is true and false at least once. By checking whether every test execution of DCC satisfies the program requirement or not, testers can achieve 100% coverage of every decision and 100% coverage of every condition in the program.

  - **Multiple Condition Coverage (MCC):** The test cases for this white-box testing technique ensure that every possible combination of conditions for every decision to be tested. Each decision with n conditions has 2^n MCC test cases. So although the test result is very precise, the cost will be really

expensive if n is very large. By checking whether every test execution of MCC satisfies the program requirement, testers can understand whether every possible combination of conditions in a decision works properly or not.

2.2 Modified Condition/Decision Coverage

This section explains MC/DC [ChM94, Whi01, HaV01] and identifies its key strengths. It also indicates what the role it plays in software testing area and finally gives the reason why it is widely used in testing avionics systems.

2.2.1 Understanding of MC/DC

The standard MC/DC definition is shown as below:

> *Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once. Each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions* .[HVC+01]

This can be summarized as follows:

Modified Condition/Decision Coverage (MC/DC) is an enhanced version of Decision Condition Coverage (DCC), it includes the basic requirements of DCC and also has its own extra requirements.

For the basic requirements part (DCC):

1) Test cases set should be able to consider all the entrance and exit situations of the program. It is the fundamental requirement of DCC and makes sure the test cases consider all the conditions and decisions in the program. This requirement is related to '*Every point of entry and exit in the program has been invoked at least once*' in the standard MC/DC definition.

2) Test cases set should be able to cause every decision to be taken all possible outcomes (true and false) at least once. This requirement satisfies '*every decision in the program has taken all possible outcomes at least once'* in the standard MC/DC definition and makes sure all possible values of every decision are under consideration.

3) Test cases set should be able to cause every condition in a decision to be true and false at least once. This requirement is related to '*every condition in a decision in the program has taken all possible outcomes at least once'* in the standard MC/DC definition and makes sure all possible values of every condition in every decision

are under consideration.

For its own extra requirement part:

4) Test cases set should make sure that each condition in a decision should affect the outcome of that decision independently. That means the outcome of the decision should be changed if only a single condition was changed. This requirement is related to the last two sentences in the standard MC/DC definition and is the kernel of MC/DC coverage criterion.

2.2.2 MC/DC Example

I have developed the following example to show how MC/DC is used. Figure 2.1 indicates a section of code in COMWS that checks whether the profile session is correct or not. This piece of code is an instance of a decision in the form A && (B || C).

```
if(session.getAttribute("ID") != null &&
    (session.getAttribute("Email") == null ||
        session.getAttribute("Password") == null)){
           result = "{'err': 'session_profile_error'}";
}
```

Figure 2.1 an instance of decision A && (B || C) in COMWS

The **truth table** of decision A && (B || C) is shown as below:

Table 2.1 truth table for A&& (B||C)

Independent Condition	A	B	C	A&&(B||C)
	T	T	T	T
B	T	T	F	T
A,C	T	F	T	T
	F	T	T	F
B,C	T	F	F	F
	F	T	F	F
A	F	F	T	F
	F	F	F	F

Test Cases

I use a list in the format {A, B, C} to indicate the MC/DC test cases for this decision:

1. {T, T, F} => T
2. {T, F, F} => F

3. {T, F, T} => T
4. {F, F, T} => F

To provide complete MC/DC test coverage on a decision of the form A && (B || C), 4 tests are required.

Explanation

A truth Table is a good way to find suitable MC/DC test cases. That's because the truth table indicates the relationship (especially the independency relationship) between conditions and decision, and can help tester to find suitable combinations of tests to meet the MC/DC requirement. Let's take the decision A && (B || C) for example, based on the truth table 2.1 shown above, the set {TTF, TFF, TFT, FFT} is a good choice. The reason is that this set meets the basic requirement 1 and 2 and also can make each condition (A, B and C) to affect the outcome of the decision independently (See the first row).

2.2.3 The Advantages of MC/DC

A comparison of different white-box testing techniques shows the advantages of MC/DC. The detailed comparisons among MC/DC, DCC and MCC together with the result are shown below:

- **Comparison on the number of tests**

  1. DCC: For a decision with N conditions, Decision Condition Coverage needs at least 2 test cases to finish the test. This is shown in Figure 2.2.

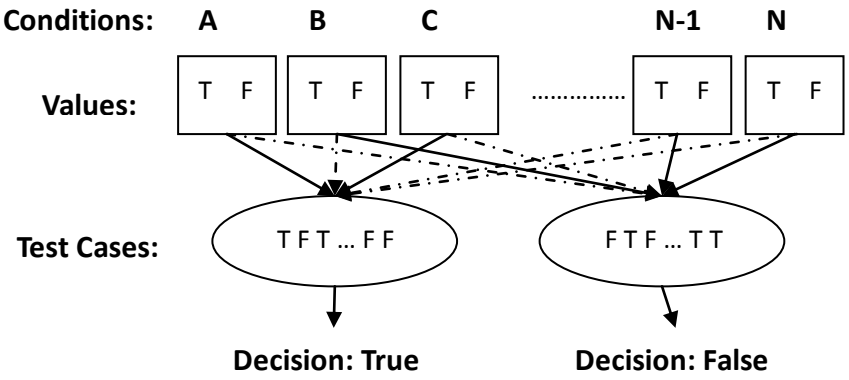


Figure 2.2 the way to obtain at least number test cases for DCC

Based on the truth table of the tested decision, one Boolean value of each condition was combined to make the decision equals to true, at the same time, the reverse Boolean value of each condition was joined to make the decision equals to false, then these two test cases can satisfy DCC requirement.

2. MC/DC: If the decision has N uncoupled conditions, then MC/DC needs at

least (N + 1) tests by varying one condition of each of the first N tests (For example: test cases {TTT, FTT, TFT, TTF} for decision (A && B && C)). If the decision has coupled conditions, then MC/DC needs at most 2N tests by selecting unique tests for each condition [ChM94]. So the range of MC/DC tests number is [N+1, 2N].

3. MCC: From the definition in 2.1, if the decision has N conditions, then MCC need 2^N test cases to complete the test.

Conclusion for this comparison: Figure 2.3 concludes the growth of test cases for DCC, MC/DC and MCC when testing a decision with different number of conditions.

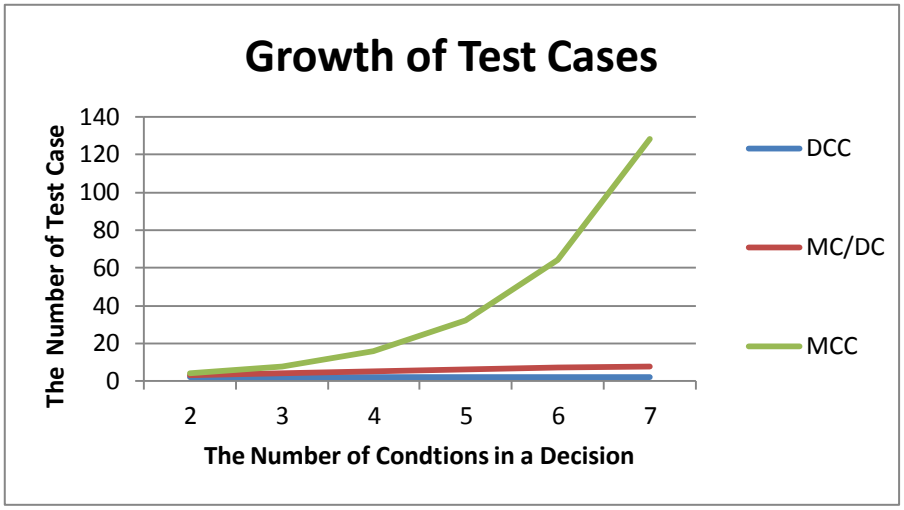


Figure 2.3 the growth of test cases for DCC, MC/DC and MCC

As Figure 2.3 shows, the number of DCC test cases has no relationship to the number of conditions in the tested decision, always stands at 2; the number of test cases for MC/DC grows slightly (linear growth) as the number of conditions in the tested decision increases. On the contrary, the number of MCC test cases increases significantly (exponential increase) with the growth of the number of tested decision's conditions.

- **Comparison on finding faults:**

  1. DCC: DCC can find many faults, but it is not very sensitive to logical operator. Table 2.2 shows such one example.

Table 2.2 DCC test cases for A &&B and A || B

Decision	DCC test cases
A && B	1. {T, T} 2. {F, F}
A || B	1. {T, T} 2. {F, F}

The DCC test cases for decision 'A && B' and 'A || B' are the same, which means

DCC is not sensitive to distinguish logical operators '&&' and '||'. So by using DCC technique, it is not easy to find such miswritten fault.

2. MC/DC: MC/DC can find more faults than DCC, that's because MC/DC focuses more on the relationship between logical operator and operands. Table 2.3 shows the same example to DCC.

Table 2.3 MC/DC test cases for A && B and A || B

Decision	**MC/DC test cases**
A && B	1. {T, T} 2. {T, F} 3. {F, T}
A || B	1. {T, F} 2. {F, T} 3. {F, F}

MC/DC is sensitive to distinguish logical operator '&&' and '||' by using the special test cases {T, F} and {F, T}, which means MC/DC has stronger ability in finding faults than DCC.

3. MCC: Although MC/DC considers the most part in testing complex decisions, it still doesn't reach the same level as MCC. MCC considers every possible combination of conditions for every decision in the tested program, so it has the strongest ability in finding faults among these three techniques.

Conclusion for this comparison: the ability in finding faults: DCC<MC/DC <≈ MCC (<≈ means smaller but similar ability)

- **Final Comparison Conclusion:**
  By combining the conclusions of these two comparisons above, the final comparison conclusion can be shown in Table 2.4.

Table 2.4 comparisons of MCC, MC/DC and DCC

Comparison Item	**DCC**	**MC/DC**	**MCC**
Number of tests for N conditions within a decision	2	N + 1	2^N
Ability in finding faults	DCC<MC/DC <≈MCC		

Table 2.4 indicates that MC/DC has a low increase in test cases when compared with **DCC** and at the same time, has a similar ability to find faults when compared with **MCC**. So when using MC/DC to test complex software, it can obtain a high accuracy testing result (finding almost all the faults) by only writing acceptable number of tests.

2.2.4 MC/DC in Software Testing

Software has become the medium of choice for enabling advanced automation in aircraft, and also in ground and satellite-based systems that manage communication,

navigation, and surveillance for air traffic control. As the capability and complexity of software-based systems increases, so does the challenge of verifying that these systems meet their requirements, including safety requirements. For systems that are safety and mission critical, extensive testing is required. However, the size and complexity of today's avionics products prohibit exhaustive testing [HaV01].

The RTCA/DO-178B document Software Considerations in Airborne Systems and Equipment Certification [HaV01, RTC92] is the primary means used by aviation software developers to obtain Federal Aviation Administration (FAA) approval of airborne computer software [HaV01, USF93]. DO-178B describes software life cycle activities and design considerations, and enumerates sets of objectives for the software life cycle processes. For level A software (that is, software whose anomalous behavior could have catastrophic consequences), DO-178B requires that testing achieve MC/DC of the software structure. MC/DC criteria were developed to provide many of the benefits of exhaustive testing of Boolean expressions without requiring exhaustive testing [HaV01, ChM94].

In the context of DO-178B, MC/DC serves as a measure of the adequacy of requirements-based testing-especially with respect to exercising logical expressions. In that regard, MC/DC is often used as an exit criterion (or one aspect of the exit criteria) for requirements-based testing [HaV01].

2.2.5 MC/DC on Avionics Systems

A more detailed analysis of avionics systems shows the fundamental reasons why MC/DC is a suitable technique for testing these systems. Avionics systems typically have a large number of complex Boolean expressions. Table 2.5 shows the number of Boolean expressions with n conditions for all of the logic expressions taken from the airborne software (written in Ada) of five different Line Replaceable Units(LRUs) from level A systems [HVC+01, Chi01].

Table 2.5 Boolean Expression Profile for 5 Line Replaceable Units

	Number of Conditions, n									
	1	2	3	4	5	6-10	11-15	16-20	21-35	36-76
Number of Boolean expressions with n conditions	16491	2262	685	391	131	219	35	36	4	2

Table 2.5 shows that the number of complex expressions is really large and these systems even have more than 36 conditions Boolean expressions. It is difficult to test them by using MCC (36 conditions need 2^{36} tests), but for MC/DC, it is not the case. Two reasons are listed below:

- As shown in section 2.2.3, MC/DC can achieve many of the benefits of MCC testing while has a low increase in required test cases.

- The other subtle reason: avionics systems are typically example of state-based systems and MC/DC has a good performance when testing these systems. When designing avionics systems, programmers have to consider a lot of factors such as height, weight, high atmospheric pressure, humidity level and so on. These factors are actually conditions inside complex expressions and the combination of different values of these factors is related to the different state of the running system.

  It is important to make sure that every factor inside the system works properly because even a small change on a factor may lead the system from good state to disastrous state. Fortunately, by using the technique MC/DC, this requirement could be satisfied. The essence of MC/DC is that each condition should affect the outcome of the decision independently, which means MC/DC can help testers to check whether each condition in each decision in the source code has the proper effect or not (Whether the changes on each factor can lead the system to correct state or not). So MC/DC is really suitable for testing those complex stated-based systems.

2.3 Problem Analysis

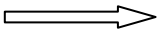
A similar approach, as used for avionics systems, is used to analyze the web server. Table 2.6 indicates the number of Boolean expressions with n conditions for all of the logic expressions taken from the COMWS.

Table 2.6 Actual Boolean Expression Profile for COMWS

	Number of Conditions, n					
	1	2	3	4	5	6
Number of Boolean expressions with n conditions	138	11	3	2	0	0

As the table data shows, COMWS does not have many complex Boolean expressions. But from an analysis of the COMWS source code, this web server has many special code structures as indicated in Figure 2.4 (one piece of such style code in COMWS is shown in Figure 2.6). Such code is named as the **Equivalent complex Boolean expression** shown in Figure 2.5.

```
if(A){
   if(B){
      if(C){
           …
       }
   }
}
```



```
if(A && B && C){
        …
}
```

Figure 2.4 special structure codes

Figure 2.5 Equivalent complex Boolean expressions

```
if(p.GetUserName().equals("") || p.GetPassword().equals(""))
{
    jsondata = "{'result': -2}"; // wrong format
}
else
{
    if(p.UserExist("")) //have exist this provider
    {
        jsondata = "{'result': 0}";
    }
    else
    {
        p.SetPassword(Encode.createPassword(jsonObject.getString("password")));
        int id = p.Insert();
        if(id == -1) //some error
        {
            jsondata = "{'result': -1}";
        }
        else
        {
            jsondata = "{'result': 1, 'id': " +id+ "}"; //should add username
        }
    }
}
```

Figure 2.6 one piece of such structure code in COMWS

After the equivalent substitution for all such code structures in COMWS, the equivalent complex Boolean expression profile for this web server is shown in Table 2.7 and the comparison between the percentage of number of Boolean expressions with n conditions in COMWS and avionics systems is shown in Figure 2.7.

Table 2.7 equivalent complex Boolean expression profile for COMWS

	Number of Conditions, n								
	1	2	3	4	5	6	7	8	9
Number of Boolean expressions with n conditions	77	19	7	6	1	1	1	0	1

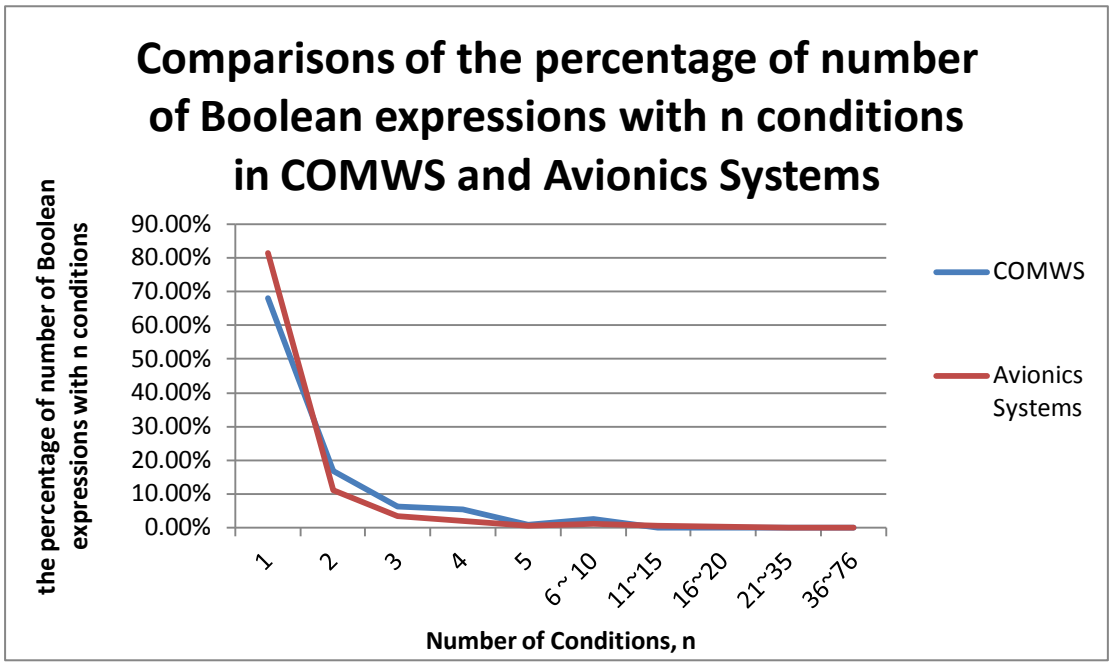


Figure2.7 comparisons of the percentage of number of Boolean expressions with n conditions in COMWS and Avionics Systems

At the same time, in order to show the complex Boolean expressions distribution difference between Avionic systems and other Java programs, another analysis is performed on Qualitas Corpus[1]. The Qualitas Corpus is a *curated* collection of Java software systems intended to be used for empirical studies of code artifacts. 14 different software systems together with their class file in Qualitas Corpus are randomly selected to be analyzed (the technique used to calculate the number of Boolean expression with n conditions for these different software systems is shown in Appendix B), and the result is shown in Figure 2.8 (The reason why the number of conditions starts with 2 is because for almost all the software systems, the number of decision with 1 condition is substantial larger than the others, so if this diagram starts with 1 condition, the complex Boolean expressions distribution difference for this software is hard to distinguish).

[1] Qualitas Corpus: http://www.qualitascorpus.com/

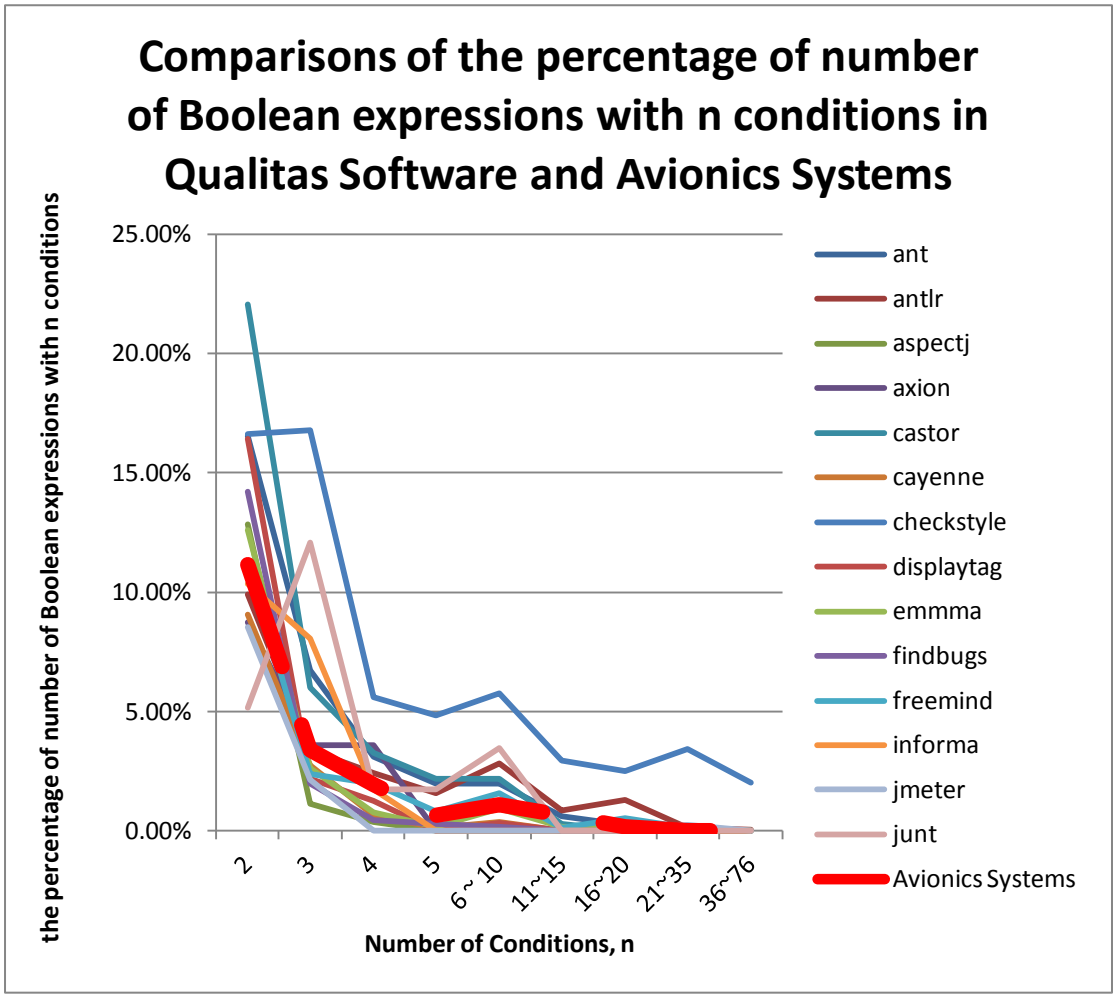


Figure2.8 comparisons of the percentage of number of Boolean expressions with n conditions in Qualitas Software and Avionics Systems

As Figure 2.7 and Figure 2.8 shows, different software has different complex Boolean expressions distribution. COMWS has a very similar distribution pattern of complex Boolean expressions to Avionics Systems (COMWS even has more the percentage of complex Boolean expressions with more than 1 condition than that of Avionics Systems). At the same time, MC/DC seems suitable to test other Java programs, it will be explored in 'Conclusions' chapter. Based on the analysis in section 2.2.5, MCC is also not a good candidate for testing COMWS.

In common with Avionics Systems, web servers are also state-based systems. They have to identify the correct next state based on the current state and user's input. For example, for the update user's profile method in COMWS (the code is shown in Figure 2.6), the web server should select the unique 1 of 4 states (-2 -> wrong format, 0 -> exist user's email, 1-> update succeed, -1 -> update failed) to identify which one should be responded to the front-end web application.

In conclusion, based on the two points above (MCC not well suited and both state-based systems), **a better test technique to test web servers is MC/DC.** It is not

only because MC/DC can achieve approximately the same level of effectiveness as MCC with significantly fewer test cases, but also the reason that MC/DC can check whether each condition in each decision in the source code has the proper effect in order to ensure the web server can jump to the correct state.

2.4 Previous Work

This section provides an overview of previous work. The previous work is divided into two parts: literature-based theory work and software-based real testing work. Literature-based theory work focus on the research on MC/DC while software-based real testing work focus on evaluating the effectiveness of MC/DC in testing the avionics related software.

2.4.1 Literature-Based Theory Work

MC/DC is a white-box testing technique in software testing area, many scientists and academic institutions have researched on it.

- **Characteristics of MC/DC:**

(1) In an investigation of the applicability of MC/DC to software testing, the modified condition/decision coverage criterion together with its properties and areas for further work were described [ChM94]. It first described the definition of MC/DC, its properties and relationship to other criteria. Then it identified problems posed by coupled conditions and the Ada short-circuits operators and proposed approaches for each. Finally, an analysis of comparing the sensitivity of the MC/DC with that of decision, condition/decision and multiple-condition testing in detecting errors in Boolean expressions was proposed. The conclusion was "*The MC/DC was shown to be significantly better than either the decision or condition/decision criterion and to compare favorably with the often impractical multiple-condition criterion.*" [CHM94]

(2) In a study of finding how good a randomly selected test-set is in exposing errors according to a given test criterion, an empirical evaluation of effectiveness of three main control-flow test criteria (DC (Decision Coverage), FPC (Full Predicate Coverage), MC/DC) has been performed [KaB03]. The study was based on exhaustive generation of all possible test-sets against ENF, ORF, VNF and ASF faults. It also analyzed the variation in effectiveness of test criteria (which is used to determine its fault detection reliability) and the influence of the number of conditions on average fault detection effectiveness and standard deviation of effectiveness for each test criteria.

The experiment results indicated that:

(1) MC/DC was found to be effective in detecting faults, but its test-sets had a large variation in fault detection effectiveness for ASF faults.
(2) The average effectiveness for DC, DC/R (DC/R is a variation of DC that is used to ensure that it is the test-set property that influences the effectiveness and not the test-set size) and FPC was found to decrease with the increase in the number of conditions, while that of MC/DC remained almost constant.
(3) The standard deviation of the effectiveness showed variation in DC, DC/R and FPC but remained constant for MC/DC.

Thus, the conclusion of this experiment is: "*MC/DC criterion is more reliable and stable in comparison to DC, DC/R and FPC*". [KaB03]

- **Test-Suite Reduction for MC/DC:**

(1) In an investigation of test-suite reduction and prioritization problem for MC/DC, new algorithms for test-suite reduction and prioritization were proposed that can be tailored effectively for use with MC/DC [JoH01]. Test suite always grows when the tested software is modified (Because new test cases will be added into test suite). Researchers developed test-suite reduction algorithms and test-suite prioritization algorithms to solve test-suite size problem. But existing reduction and prioritization techniques may not be effective in reducing or prioritizing MC/DC-adequate test suites because they do not consider the complexity of the criterion. Because of this, new algorithms were proposed and some evaluation was performed on these two algorithms. The evaluation conclusion was "*The algorithms show the potential for substantial test-suite size reduction with respect to MC/DC" and "the test-suite prioritization efficiently orders a test suite for MC/DC*". [JoH01]

(2) In an investigation of test-suite reduction problem for MC/DC, a new test-suite reduction technique that using bi-objective model for MC/DC was proposed to address this problem [PZL+05, PMG+10]. MC/DC is an effective verification method that can help to detect safety faults, but its cost is a little bit expensive. In regression testing, it is quite costly to rerun all of test cases in test suite. Therefore, it is necessary to reduce the test suite to improve test efficiency and save test cost. However, many existing test-suite reduction techniques are not effective to reduce MC/DC test suite. Because of this, a new algorithm was presented that has two characteristics from traditional reduction algorithm:
(1) Constructing a bi-objective model that considers both coverage degree and fault-detection ability of test cases
(2) The reduced test suite satisfies adequate coverage for test-requirement suite no matter when the algorithm is terminated.

Some experiments were performed on this algorithm and the results indicated that "*this test-suite reduction algorithm is effective for both reducing test suite and ensuring fault-detection ability of reduced test suite.*" [PZL+05]

- **Variants of MC/DC:**

(1) Observable Modified Condition/Decision Coverage (OMC/DC) [WGY+13]
Structural coverage metrics, such as MC/DC, are commonly used to measure the adequacy of test suites. Such criteria require only certain code structures (such as a particular Boolean assignment of a decision) to be exercised, without requiring the resulting value to affect an observable point in the program. As a result, test suites satisfying these criteria can produce corrupted internal state without revealing a fault, resulting in wasted testing effort. In order to solve this problem, Observable MC/DC, a combination of traditional MC/DC testing with a notion of observability (an additional path constraint) was proposed to help ensure that faults will be observed through a non-masking path from the point the obligation is satisfied to a variable monitored by the test oracle. Some experiments were performed to test OMC/DC and the result indicated that "*OMC/DC test suites locate a median of 17.5% more faults than MC/DC test suites when paired with an oracle observing only the output variables*" and "*OMC/DC is less sensitive to the structure of the program under test than MC/DC and also provides the benefits of using a very strong test oracle with MC/DC coverage*". [WGY+13]

(2) Reinforced Condition/Decision Coverage (RC/DC) [ViB06]
The MC/DC criterion is used mainly for testing of safety-critical avionics software. The main aim of MC/DC is testing situations when changing a condition implies a change in a decision. But this criterion has one substantial shortcoming - the deficiency of requirements for testing of the false actuation (The false actuation of a system could be invoked by a software error in situations when changing a condition should not imply changing a decision) type of failures. Such situation could make MC/DC insufficient for many safety-critical applications. To eliminate this shortcoming, a new criterion RC/DC was proposed. Testing according RC/DC should include test cases according MC/DC and additional test cases for testing important situations when a false actuation of a system is possible. In that way, all requirements of MC/DC are valid and a new requirement for keeping the value of a decision when varying a condition is added to the testing regime. RC/DC can make the testing process more effective and it could be important in the testing of safety-critical applications.

2.4.2 Software-Based Real Testing Work

(1) An empirical evaluation of MC/DC coverage criterion was performed on the HETE-2 Satellite Software [DuL00]. In order to be certified by the FAA, aviation software must satisfy a standard labeled DO-178B and the most critical (Level A) software must satisfy MC/DC. But this standard is controversial in the aviation community, partially because of perceived high cost and low effectiveness. In order to shed some light on this issue, an evaluation was performed on the attitude control software (ACS) of the HETE-2(High Energy Transient Explorer) scientific satellite being built by the MIT Center for Space Research for NASA.

Functional testing and functional testing augmented with test cases were compared to satisfy MC/DC coverage on the ACS. For black-box testing part, three-step testing process was executed for each mode: switching logic testing, parameter testing and functional testing. For white-box testing part, the coverage evaluation was firstly performed to identify the parts of the code that have been left unexplored by black-box testing and then the MC/DC additional test cases were designed to find additional errors.

After the experiment, two points were summarized:

(1) The test cases generated to satisfy the MC/DC coverage requirement detected important errors not detectable by functional testing. These additional tests corresponded to four kinds of limitations of the black-box testing process:
- Something was forgotten during black-box testing.
- The software has a complex logic mechanism requiring in-depth understanding and precise, customized testing.
- Some feature of the software was not included in the specification and could not give rise to a test case in a black-box testing context.
- The effects of some errors were too small to be detected by black-box testing.

(2) Although MC/DC coverage testing took a considerable amount of resources (about 40% of the total testing time), it was not significantly more difficult than satisfying condition/decision coverage and it found errors that could not have been found with that lower level of structural coverage.

(2) An evaluation of MC/DC testing was performed on an industrial PLC logic networks [Bis03]. Coverage measurement has been used to estimate residual faults in program code. The same concept can also be applied to PLC logic networks. Other than the normal coverage concept, a PLC logic network has its equivalent logic coverage - input-output pair coverage (I-O pair coverage), which means that input values are selected such that change of a given binary input i can "toggle" the state of a binary output j. There is a strong relationship between

input-output pair coverage and the MC/DC test method, so it is really deserve to research whether MC/DC can maximize such coverage growth and improve fault detection efficiency or not on a PLC logic network.

The PLC logic network used in this paper was taken from an industrial example that had 36 binary inputs and 10 binary outputs. There were also 6 known faults in the initial logic implementation. For the experiment, I-O pair coverage was compared against faults found using three different test strategies: MC/DC random testing, uniform random input testing and a set of 486 systematic tests developed for the original industrial logic implementation. The experiment result indicated that the MC/DC test appears to out-perform the other rest strategies, finding the first 4 faults in 10 random tests and all 6 in 486 tests, while the systematic tests had only detected 4 faults at this stage, and random input testing had found none.

After the experiment, two conclusions were summarized:

(1) I-O pair coverage was strongly correlated with the faults found in the logic network.
(2) MC/DC random testing was more effective than random input testing and an existing systematic test set-probably because coverage growth was faster.

2.5 Proposal

The similar pattern of complex Boolean expressions as shown in section 2.3 indicates that MC/DC is a suitable coverage criterion for testing web servers. The proposal of this project is to evaluate the effectiveness of MC/DC for testing the COMWS. In order to implement this task, four steps were proposed:

1) Select representative java files

Figure 2.9 below shows the java files list about the COMWS. Those important functional-related java files were selected (such as CultureObject_Interface.java, SymbolTransfer.java and so on, detailed function description of COMWS is shown in Chapter 4) to do the experiment in order to obtain useful evaluation data.

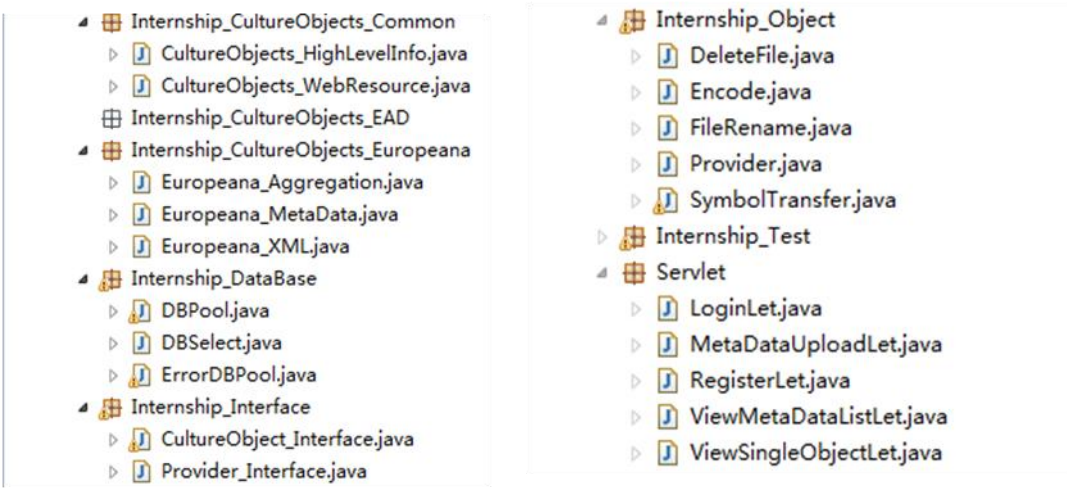


Figure 2.9 java file list of COMWS

2) Implement the Black-Box testing on selected java files

Three Black-Box testing techniques (Equivalence Partitioning, Boundary Value Analysis and Combinations of Inputs) were used in testing each method in each selected java files. The items listed below should be calculated after the Black-Box testing:

1. Black-Box tests number
2. The work (cost time) to write these Black-Box tests
3. Faults found number
4. Branch coverage of tested program
5. Code coverage of tested program

3) Implement the MC/DC testing on selected java files

First generated the test cases based on the equivalence conditions mentioned in section 2.3 and truth table for each decision and then wrote and executed the tests to do the MC/DC testing. The items listed below should be calculated after the MC/DC testing:

1. MC/DC tests number
2. The work (cost time) to write these MC/DC tests
3. Faults found number
4. Branch coverage of tested program
5. Code coverage of tested program
6. Extra MC/DC tests number to find faults which BB testing didn't find

4) Compare Black-Box testing results with MC/DC testing results and draw conclusions

Compared the Black-Box testing with MC/DC testing based on the calculated

items list above and focused on the following three criteria to draw the conclusion:

- Testing cost:
  Comparison was based on two areas: the number of tests and the time taken to write tests for Black-Box testing and MC/DC testing
- Faults found:
  Focused on whether MC/DC found extra faults when compared with Black-Box testing. If it was, counted how many MC/DC tests were executed to find these extra faults.
- Program coverage:
  Comparison was based on two areas: branch coverage and code coverage of the tested programs.

3. Software & Tools Used

This section provides a brief description of the software and tools that were used in the implementation and testing of the COMWS.

3.1 Tomcat and MySQL

Tomcat [CBE+04] is an open source web server and servlet container, it implements the Java Servlet and the JavaServer Pages(JSP) specifications from Sun Microsystems, and provides a 'pure Java' HTTP web server environment for Java code to run in.

MySQL [MyS01] is an open-source relational database management system (RDBMS) that runs as a server providing multi-user access to a number of databases, it is always used in web application development and can be accessed and managed by some GUI tools such as phpMyAdmin, DBEdit and so on.

In this project, Tomcat was used as the web server and MySQL was used as the way to store users' and cultural objects' information. Meanwhile, phpMyAdmin (as shown in Figure 3.1) was used as the GUI tool to manage the MySQL database (such as create database, modify the structure of different tables and view detailed data in order to check whether the related operation is correct or not).

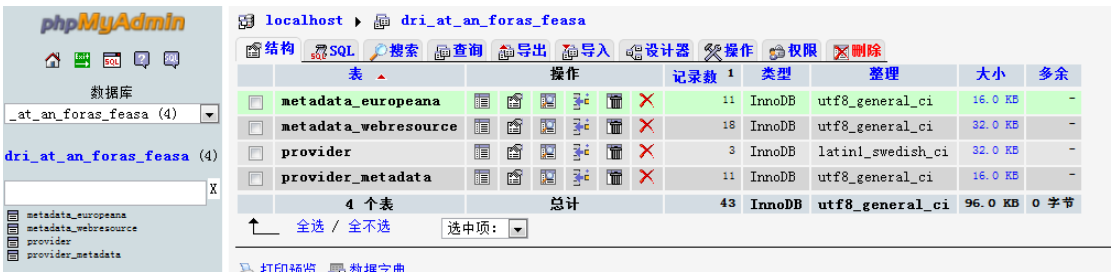


Figure 3.1 phpMyAdmin to manage project database

3.2 Customized JUnit Test Runner

When doing the unit testing and MC/DC testing of the COMWS, instead of viewing where was wrong inside the code, detailed testing results and the reason why errors occurred should be collected. The traditional JUnit cannot satisfy this requirement, so customized JUnit test runner should be implemented to run those test cases.

Implementing customized JUnit test runner includes two steps [BTL+11]:

1. Define my own annotation (as shown in Figure 3.2)

```
1 package Internship_Test;
2
3 import java.lang.annotation.*;
4 @Retention(RetentionPolicy.RUNTIME)
5 @Target(ElementType.METHOD)
6 public @interface MyTest { }
7
```

Figure 3.2 my own annotation

The 'retention policy' makes sure that the annotation is kept at runtime, the 'target' means the annotation is used on a method and the 'interface' indicates the name of the annotation is 'MyTest'. By defining my own annotation, test cases can be annotated by this new annotation so that test runner can find and execute them to collect test results.

2. Write the test runner to find and execute the annotated methods from a class (as shown in Figure 3.3)

```
public class MyTestRunner {

    private int runs = 0;
    private int successes = 0;
    private int fails = 0;
    private int executionFailures = 0;
    static String reason;

    void runTests(String sutName){
        try {
            for (Method m:Class.forName(sutName).getMethods())
                if (m.isAnnotationPresent(MyTest.class)) {
                    runs++;
                    if((Boolean)m.invoke(null)) {
                        successes++;
                    }
                    else
                    {
                        fails++;
                        System.out.println("Test "+m.getName()+" failed: "+reason);
                    }
                }
        }catch (Throwable ex) {
            System.out.println(ex.toString());
        }
    }
```

Figure 3.3 customized JUnit test runner

The most important method in customized JUnit test runner is runTests(), which accepts the name of a test class as its parameter, finds the test methods (annotated with 'MyTest') by using Java Reflection and execute these test methods by calling invoke().
The test results are stored in the variables *runs*, *successes*, *fails* and *executionFailures* and the detailed failure reason is shown in the variable *reason*.

By using customized JUnit test runner, my own format of test results can be generated and the detailed failure reason can be viewed through the *reason* variable. What's more, the test results can be saved to a log file so that these data can be used in the later analysis and evaluation phase.

3.3 Eclemma

Eclemma [Hof11a, Hof11b] is an open-source tool for measuring and reporting code coverage of Java programs. Testers always focus on two aspects to analyze the coverage result:

1. Coverage overview:
As shown in Figure 3.4, the coverage view lists coverage summaries for the tested project, including the information about 'coverage ratio', 'Items covered', 'Items not covered' and 'total items'. This coverage view provides a brief and clear way for testers to understand the basic statement coverage of java programs – which line is executed and which line is not.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Encode.java	0.0 %	0	152	152
DeleteFile.java	0.0 %	0	129	129
SymbolTransfer.java	92.7 %	343	27	370
SymbolTransfer	92.7 %	343	27	370
main(String[])	0.0 %	0	14	14
JsonToString(String)	94.9 %	185	10	195
isDigit(char)	100.0 %	10	0	10
isHexDigit(char)	100.0 %	10	0	10
isHexLetter(char)	100.0 %	16	0	16
isLetter(char)	100.0 %	16	0	16

Figure 3.4 coverage overview

2. Source highlighting:
As shown in Figure 3.5, the result of a coverage session is also directly visible in the Java source editors. A customizable color code highlights fully, partly and not covered lines (green - fully covered lines, yellow - partly covered lines, red - lines that have not been executed at all). So by analyzing this source highlighting, testers can understand the statement coverage and branch coverage of the tested Java program.

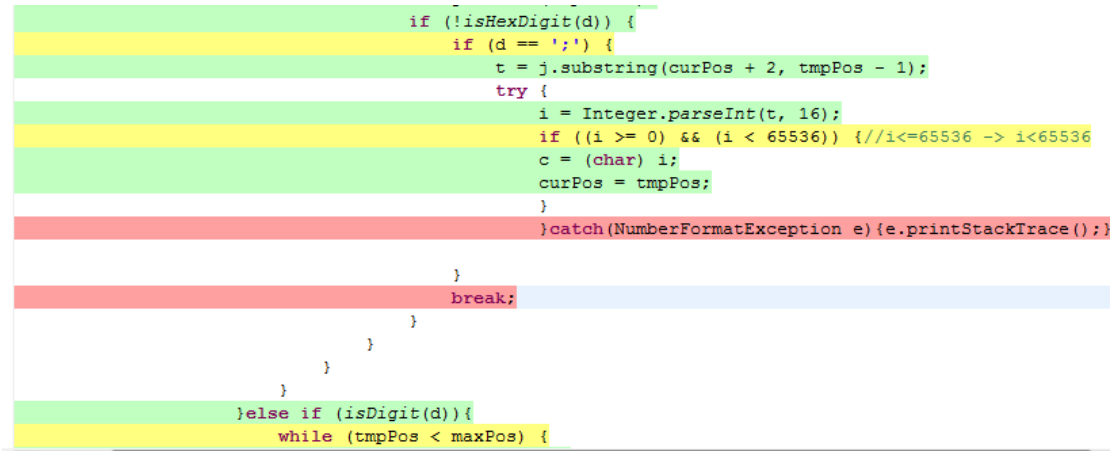


Figure 3.5 sources highlighting for coverage

4. COMWS Overview

This section focuses on the overview of COMWS. It describes what the COMWS is and shows what functions this web server has. It also gives the class diagram of COMWS to show which classes are important and to be worth testing.

4.1 Cultural Objects Management Web Server

COMWS is a web server which is used to help users to manage their cultural objects in different data formats (such as Europeana [DGH+10], DC [Dub03]) and acts as intermediate software to connect with large repositories to do cultural objects data exchange. COMWS is the core of the whole Cultural Objects Management Project (COMP). To convey an abstract overview of COMWS, the COMP architecture diagram (To show what role the COMWS plays in COMP) and COMWS Use Case diagram (To show what functions the COMWS has) are presented in section 4.1.1 and 4.1.2.

4.1.1 COMWS in COMP

The architecture of COMP is shown as below:

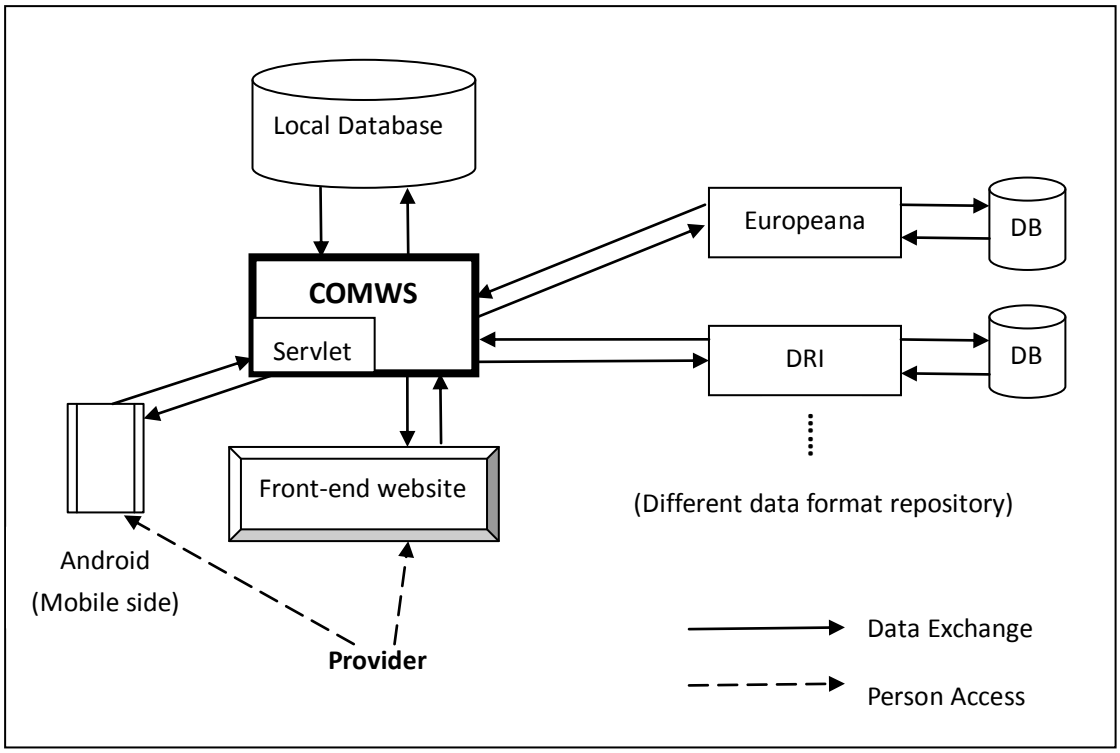


Figure 4.1 the architecture of COMP

Figure 4.1 indicates that the COMWS plays three roles in COMP:

1. Acting as the back-end server to communicate with the front-end website. When user uses the website to manage cultural objects, all the back-end operations such as insert new cultural objects into database, count user's current cultural objects number are done by COMWS.
2. Using Servlet to communicate with mobile side. User can use the mobile app to communicate with COMWS through Servlet. It helps user to record and upload their cultural objects information rapidly.
3. Doing data exchange with different data format repository. COMWS can generate data format-related storing file (for example, generating Europeana XML file of cultural objects so that it can be collected by Europeana staff) and obtain data information from these repository.

4.1.2 User Case Diagram

The COMWS's UML Use Case diagram is shown in Figure 4.2 to manifest what users will be able to do with this server.

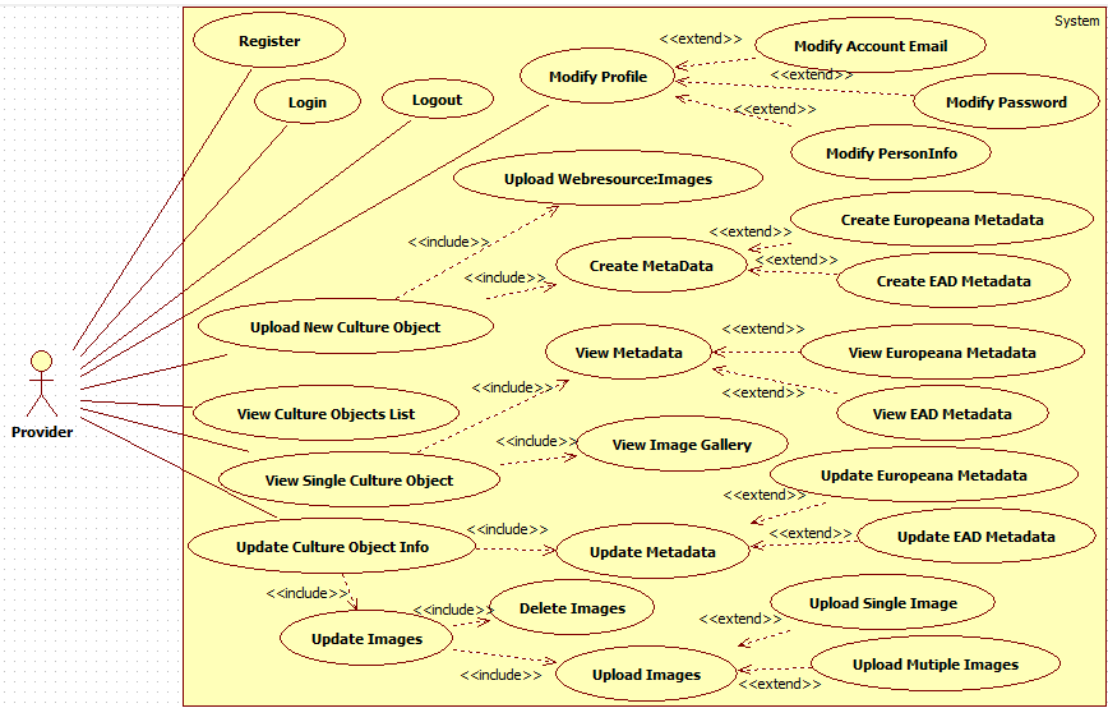


Figure 4.2 COMWS use case diagram

These use cases help to convey the functionality provided by COMWS.

4.1.3 COMWS Functionality Summary

Based on the user cases above, a summary of the entire functionality of COMWS is presented as below:

☐ **Provider's Account and Personal Info Part:**

- Register: Focus on doing the user's register operation, including check user's input register information, store information to database and return back the register result.
- Login: Focus on connecting with database to check the user's login information (email and password) in order to identify whether user can login the system or not
- Modify: it has three separate parts.

  1. Modify personal info: check the reasonableness of modified username or phone number value, update information in database and return back modify result.

  2. Modify account information: check the reasonableness of modified email value (format and whether it is duplicate or not), update information in database and return back modify result.

  3. Modify password (safety part): check the reasonableness of modified password value, update information in database and return back modify result.

- **Provider's Cultural Objects Part:**
  - Upload new cultural object:

    1. Create Metadata: receive metadata from user's side and store them into corresponding table in the database based on the data format selected by user.

    2. Upload Images: receive images from user's side; store them in the cultural object-related folder and insert images' link into the database.

  - View cultural objects list:

    Receive user's requirement and search from the database to select user's current cultural objects, return these objects information back to the front-end

  - View single cultural object:

    1. View Metadata information: receive user's different data format request, search the database and send back the corresponding metadata information of the selected cultural object to the front-end

    2. View Images: receive user's request and search all images' link of selected cultural object in the database and send back them to the front-end

  - Update cultural object info:

    1. Update metadata info: receive user's request (including the selected data format and update data), update the database and send back the result to the front-end

    2. Update images info:

    For delete image: receive user's request about which image should be deleted, remove the selected image from the folder and delete the record in the database

    For upload single image: receive user's request, store the new image into the correct folder and insert this image record into the database

    For upload multiple images: receive user's request, store all the new images into the correct folder one by one and insert these images record into the database

- **Generating Data-Format Related File Part:**

  Based on the data-format, generating corresponding file (For example, Europeana XML file) and store it into a selected folder.

4.2 COMWS Classes Overview

The class diagram of COMWS is shown in Figure 4.3.

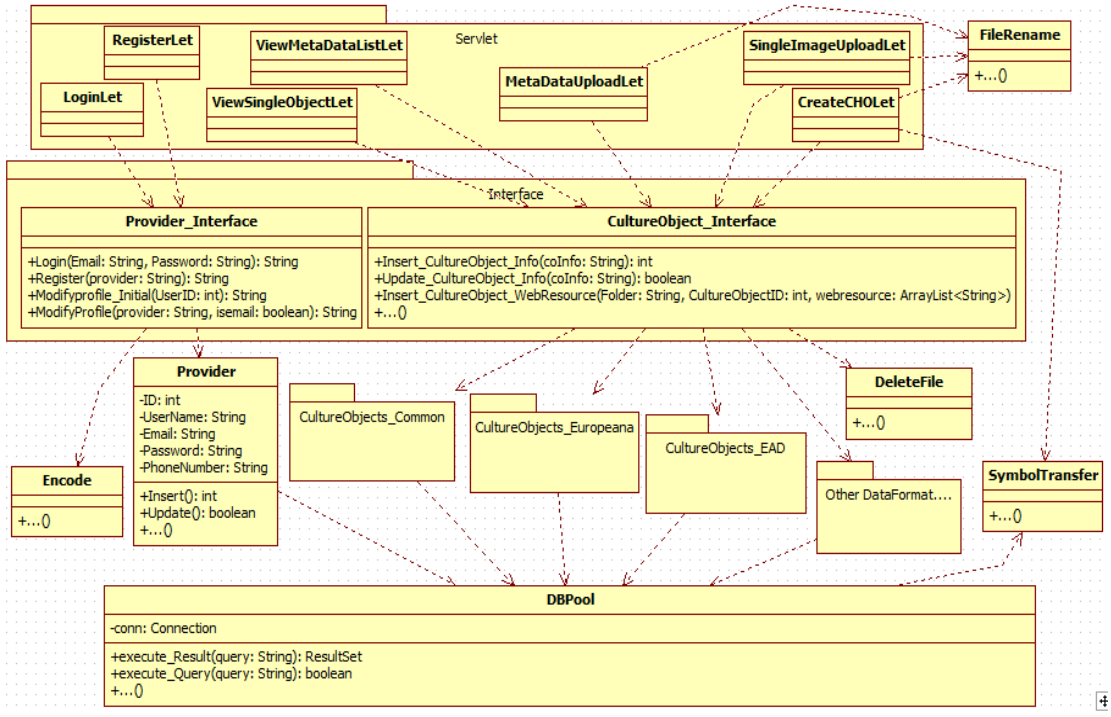


Figure 4.3 Class Diagram of COMWS

As shown in Figure 4.3, COMWS has four main levels: Database Operation Level, Objects Level, Interface Level and Servlet Level.

- Database Operation Level: DBPool class is acted as the bottom level of COMWS. It provides the database connection, executes the requirements from high level and return back data or operation result.

- Objects Level: this level contains three parts: Provider, CultureObjects and RelatedFunction.
  - Provider:
    The Provider class builds the connection between database and Provider objects. It includes the basic data storage functions, such as insert the provider's personal information into database, modify provider's email address and so on.
  - CultureObjects:
    For a single cultural object, it contains two types of information:
    1. Common information: it records web resources (images, videos and so on) and ownership information of the cultural object. Figure 4.4 indicates the package and classes used to represent such information.

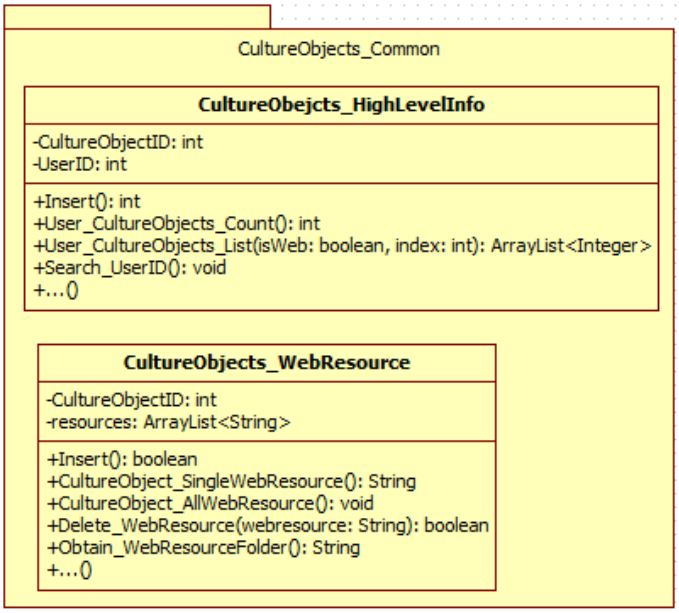


Figure 4.4 Common Information

The class CultureObjects_HighLevelInfo is used to handle ownership problem of the cultural object, such as search the cultural object belongs to which user, obtain a user's cultural objects list and so on. The class CultureObjects_WebResource manages the web resources of the cultural object and makes sure the web resources storage process is correct.

2. Metadata information: user can use different metadata types (Europeana, EAD and so on) to represent the cultural object. Each metadata type has its own data format and related functions. For example, Figure 4.5 indicates the package and classes used to represent Europeana metadata type and related functions.

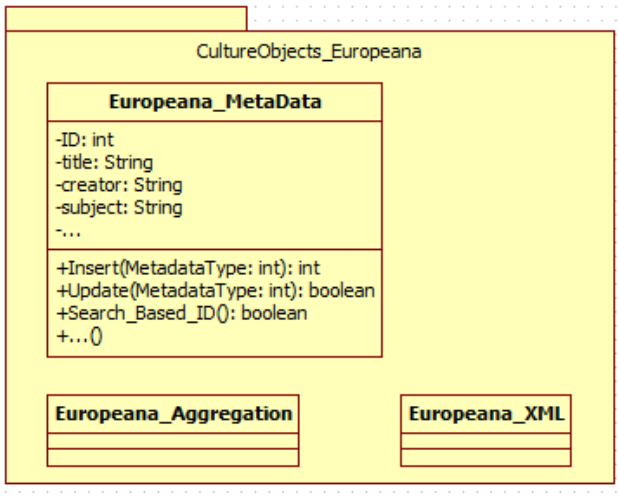


Figure 4.5 Europeana Related Metadata and Functions

The class Europeana_MetaData represents the Europeana data format to describe the cultural object. Both the classes Europeana_Aggregation and Europeana_XML are used in the cultural objects data exchange between COMWS and Europeana repositories.

- RelatedFunction:
  This part includes those classes that are used to help COMWS works properly.
  1. Encode Class: it is used to encrypt and decrypt user's password in order to improve the safety of COMWS
  2. DeleteFile Class: it is used to delete user's web resources on the server
  3. SymbolTransfer Class: it is used to handle some special symbols such as ', ", \r\n and so on.
  4. FileRename Class: it is used to rename user's upload files

- Interface Level: this level contains two interfaces: Provider_Interface and CultureObject_Interface. Provider_Interface provides the services that are related to user's operation, such as login, register and so on. CultureObject_Interface provides the services that are related to cultural objects, for example, insert web resources of cultural object, insert metadata info of cultural object and so on.

- Servlet Level: this level is used to provide services to mobile application. All the servlet classes will call Provider_Interface and CultureObject_Interface to perform their tasks.

Among these four levels of COMWS, the most important one is Interface Level. That's because it has direct relations with the main functions of COMWS, which means this level is worth to be tested.

5. Testing

This section gives a brief description of the unit testing on COMWS. It shows the experimental methodologies and gives detailed information on how MC/DC testing was designed to test this web server. It also provides examples to show the MC/DC testing processes in detail, describes the problems encountered during the implementation, and indicates how solutions work to overcome them (The detailed Black-Box testing design and implementation are shown in Appendix A).

5.1 Experimental Methodologies

When testing a web server, Black-Box testing is performed first to check whether these server methods meet their specifications or not. Such testing is the foundation

of whole web server testing (Failed to pass Black-Box testing means that the tested method is not succeed to be realized, so there is no need to do other testing) and is used to find those small and obvious faults. But Black-Box testing has shortcomings. Black-Box tests are generated based on specifications, so such tests may not be available to detect faults inside the implementation code. In order to find these faults, MC/DC was selected to provide additional support.

When measuring the effectiveness of MC/DC in testing COMWS, three criteria were used: Testing Cost, Faults Found and Program Coverage. The reasons for selecting such three criteria are:

(1) **Testing Cost**: No matter what testing technique is selected and used, its testing cost needs to be considered. If a testing technique has a strong ability to detect faults but needs huge number of tests, or takes a great deal of time, the technique is less useful in practice. For example, MCC has a strong ability to detect faults because it considers the combination of each condition inside the tested decision, but it is limited because of its incredible number of tests. For a decision with n inputs, MCC requires 2^n tests. In cases where n is small, running 2^n test may be reasonable; running 2^n tests for large n is impracticable. So the use of this criterion in this project aims to evaluate whether MC/DC was actually cost-effective in testing COMWS.

(2) **Faults Found**: this criteria directly reflects how strong a testing technique is in detecting faults in the tested program. Based on the theoretical analysis mentioned in Chapter 2, MC/DC should demonstrate a similar ability to find faults when compared with MCC. Such ability reflects that it ought to detect some faults that Black-Box testing doesn't find. So the use of this criterion in this project aims to evaluate the fault finding ability of MC/DC.

(3) **Program Coverage**: criteria code coverage and branch coverage are used to identify untested code or untested branches in the tested programs. Generally speaking, insidious faults are often found in such untested code or untested branches, so higher code coverage and branch coverage may indicate less faults remained in the tested programs. So the use of this criterion in this project aims to evaluate the program-coverage ability of MC/DC in testing COMWS (whether MC/DC can actually improve code coverage and branch coverage of the tested programs in COMWS).

    It is worth noting that the branch coverage here is not the same to traditional branch coverage. Traditional branch coverage is actually 'source code' level branch coverage. For example, Figure 5.1 shows a Boolean expression to check whether `fis` is file or not and Figure 5.2 shows its 'source code' level branch paths.

```
if(!fis.isFormField() && fis.getName().length()>0)
{
    ……
}
```

Figure 5.1 source codes to explain different branch coverage

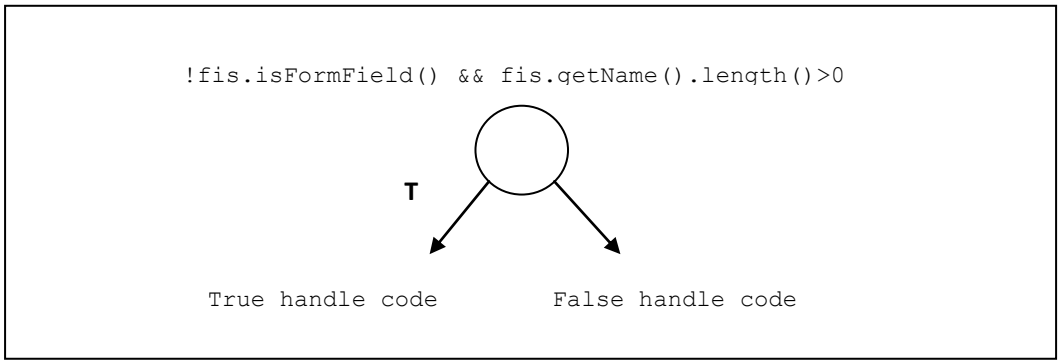


Figure 5.2 source code level branch paths

As shown in Figure 5.2, for 'source code' level branch coverage, there is no need to consider the inner conditions of the decision, just focus on whether the decision (branch) takes True or False at least once or not. Such branch coverage is not available to show the strength of MC/DC testing.

The branch coverage used in this project is actually the 'byte code' level branch coverage and the branch coverage calculated by 'Eclemma' is also such type coverage. It considers the inner conditions of the decision and checks whether each condition takes True or False at least once or not. Such coverage actually focuses on the byte code of the tested method. Each jump situation (jump succeed or not) of each jump instruction in the byte code is under consideration.

By measuring 'byte code' level branch coverage, MC/DC testing can show its strength in testing complex Boolean expression. For example, the Figure 5.3 and 5.4 indicates the 'byte code' level branch coverage results for Black-Box testing and MC/DC testing on the special complex Boolean expression in method `Login()`.

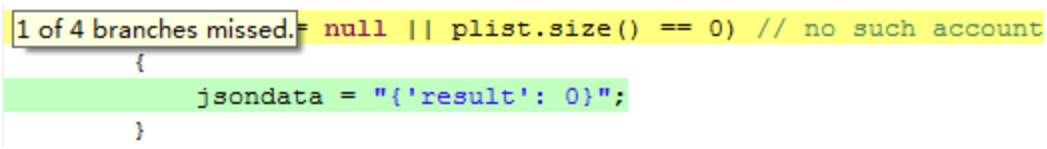


Figure 5.3 'byte code' level branch coverage results for Black-Box testing

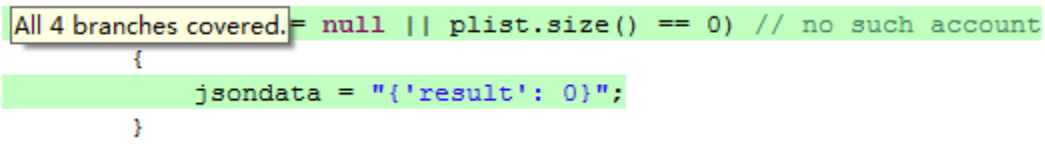


Figure 5.4 'byte code' level branch coverage results for MC/DC testing

As shown in Figure 5.3 and 5.4, Black-Box testing missed one 'byte code' level branch while MC/DC testing covered all of them. This different is due to the special code `plist == null`. Such code equals to True if and only if there is a database error inside the execution of this method. But database error has no relations to these values of input parameters, so Black-Box testing did not cover this branch. But it is not the case for MC/DC testing. One special MC/DC test case focused on this situation and the related test replaced correct database with an error-inside database to manually making a database error, which finally made this branch been covered.

By measuring 'byte code' level branch coverage, it is possible to measure the MC/DC test coverage criteria on tested programs.

The experimental methodologies above concludes the reason why Black-Box testing was performed first in testing COMWS and why used Testing Cost, Faults Found and Program Coverage (also explain the 'byte code' level branch coverage) to evaluate the effectiveness of MC/DC in testing COMWS.

5.2 Test Design

5.2.1 Test Procedure

The detailed COMWS test procedure is shown in Figure 5.5.

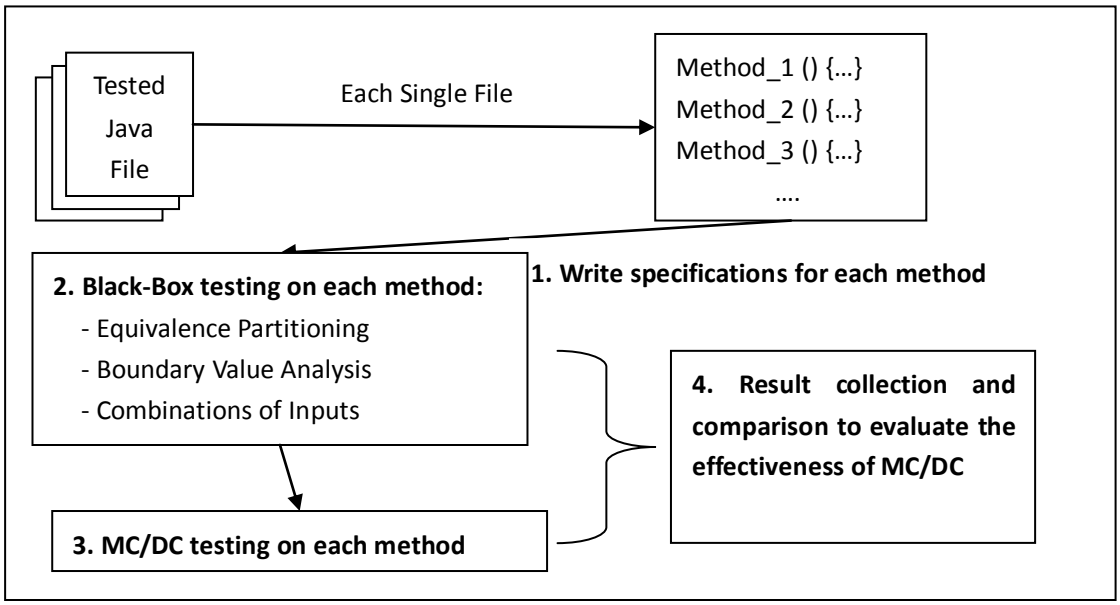


Figure 5.5 COMWS test procedure

Figure 5.5 indicates that the test procedure includes 4 steps: specifications generation, black-box testing, MC/DC testing and result collection and comparison. The detailed descriptions of these 4 steps are listed as follows.

5.2.2 Specifications Generation

Specifications play a key role in testing. In order to be tested, the correct behavior of software must be known. Specifications meet this requirement and describe both normal and error behavior of the tested software, which makes it easy for testers to design related tests [BTL+11].

In this project, for each method in each tested java file, a related specification should be given. Generating specifications for these tested methods are based on two items: specification template and COMWS functionality description.

(1) Specification template

The specification template used in this project is shown in Figure 5.6.

```
/*Specification of Method_Name():
Function:
   describe the function of this method

Program Inputs:
   parameter 1: the range
   parameter 2: the range
   ......

Program Outputs:
   return value:
   possible return value 1 - the condition 1 to cause this return value
   possible return value 2 - the condition 2 to cause this return value
   ......
*/
```

Figure 5.6 specification template

Explanation of this specification template

- The 'Function' field describes the function of the tested method and indicates the role it plays in the whole software.

- The 'Program Inputs' field describes the ranges of values for each parameter of the tested method. Understanding the ranges of values for each parameter is

really necessary because the output of the method is always affected by the values of these parameters. If the method's realization doesn't meet the parameter range requirements, then the return value of the method will not be correct, which can help testers to create tests to find these errors.

- The 'Program Outputs' field describes the ranges of the output values of the tested method. By considering this field, testers can understand the relationship between program inputs and outputs (what values of inputs will cause related outputs) so as to help them to design reasonable tests to find errors.

(2) COMWS Functionality Description

COMWS functionality description (the summery version is mentioned in section 4.1.3) gives all the detailed information about the tested methods. It includes these methods' definition, possible input value of these methods' parameters and possible output values of these methods. It indicates the role of these methods play in the whole web server.

By combining the COMWS functionality description and specification template, the specifications of all the tested methods in COMWS are generated, which makes a good preparation for the next testing stage.

5.2.3 Black-Box Testing

In black-box testing stage, for each tested method, three steps were required to check whether it meets specification requirement or not.

1. **Three Techniques Test Cases Analysis and Test Generation**
   Three techniques were used to do Black-Box testing on COMWS: Equivalence Partitioning (EP), Boundary Value Analysis (BVA) and Combinations of Inputs (CI). Different test analyses were performed on these three techniques to generate their own tests set.

2. **Test Realization**
   Based on the test data for these three techniques generated in step 1, the detailed source code for each test was realized in this stage.

3. **Test Execution**
   In this stage, tests were executed using customized JUnit test runner and the Black-Box testing results were collected.

The detailed description for each Black-Box testing step is shown in Appendix A.

5.2.4 MC/DC Testing

In MC/DC testing stage, for each tested method, a four steps procedure is required. Such test procedure is shown in Figure 5.7.

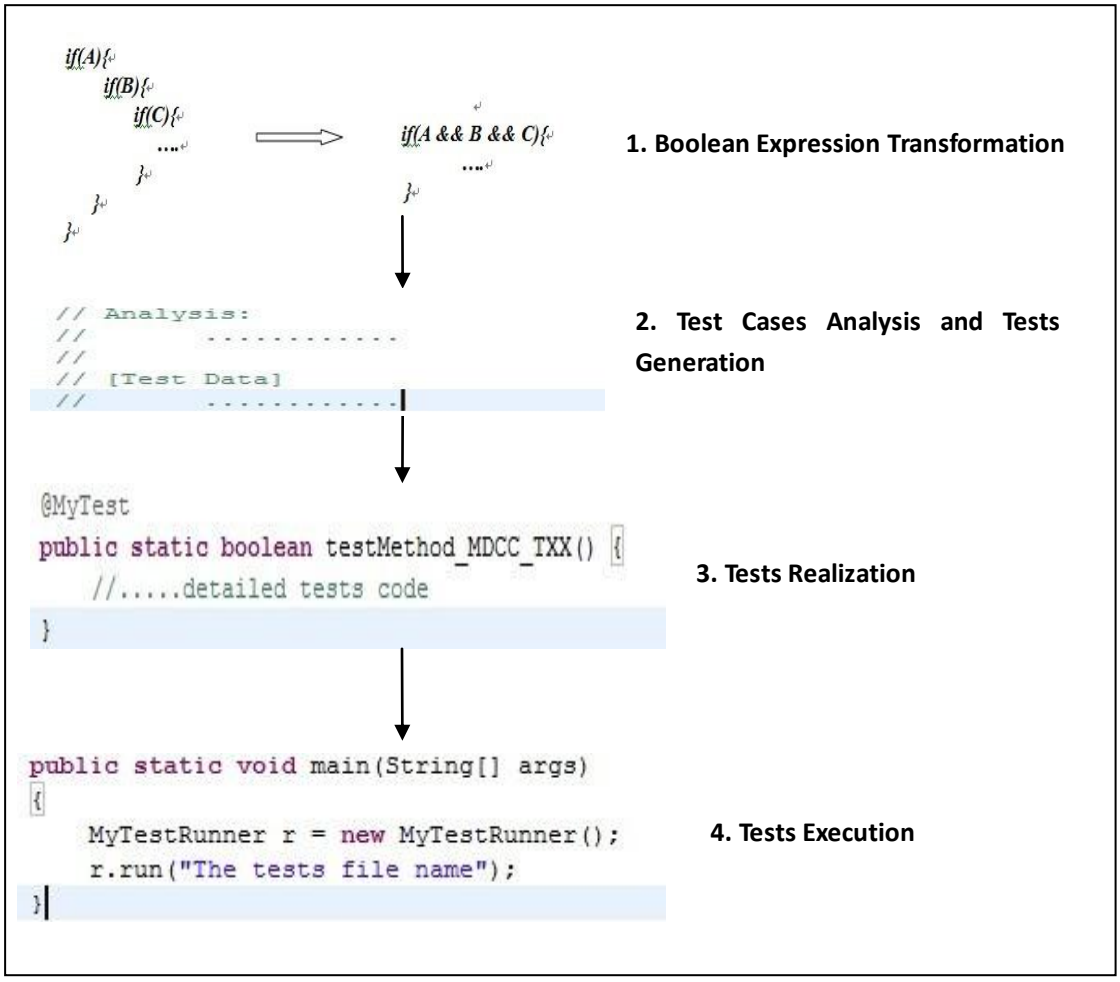


Figure 5.7 4 steps procedure for MC/DC testing

1. Boolean Expression Transformation

Because COMWS has so many such special codes mentioned in section 2.3, the first step is doing a transformation on the code of tested method to change all such codes to the equivalent complex Boolean expressions. This is the basic step to do the MC/DC testing on COMWS.

2. Test Cases Analysis and Tests Generation

The second step is doing the test cases analysis on these equivalent complex Boolean expressions and generating the related MC/DC tests. The analysis template is shown in Figure 5.8.

```
// Analysis:
//
//   Conditions:
```

```
//       A: Line xx: ...
//       B: Line xx: ...
//       C: Line xx: ...
//       .....
//
//   Equivalent Complex Boolean Expression:
//       ......
//
//   Truth Table:
//       A       B       C     ...    Complex Boolean Expression
//      ...     ...     ...    ...     ...
//      ...     ...     ...    ...     ...
//
//   Test Cases:
//      CASE    A         B         C       ...      Test
//      =====   ============================      =====
//      MCDC1   ...       ...       ...      ...      Test1
//      MCDC2   ...       ...       ...      ...      Test2
//      ...     ...       ...       ...      ...       ...
```

Figure 5.8 analysis template for MC/DC

As Figure 5.8 indicates, the conditions and equivalent complex Boolean expression fields show which conditions and decision are under consideration. The truth table gives the relationship between the output value of decision and the truth value of each condition. Then, based on the truth table, the test cases set can be generated (Selecting from the truth table to make the test cases set satisfies the MC/DC requirement) and each test case is related to one test.

Based on the MC/DC test cases generated from the analysis stage, combined with the tested method's inputs, the MC/DC test data of this method can be produced, just as shown in Figure 5.9.

```
//  Test Data:
//         Test Cases                Inputs              Expected Outputs
//    ID    Covered            xxx   xxx   xxx  ...       return value
//    Test1 MCDC1               ...   ...   ...  ...        ...
//    Test2 MCDC2               ...   ...   ...  ...        ...
//    Test3 MCDC3               ...   ...   ...  ...        ...
//    ...   ...                ...   ...   ...  ...       ...
```

Figure 5.9 MC/DC Test Data Generation

3. Tests Realization

Based on the test data generated in step 2, the detailed code for each test can be realized in this stage. The code template of MC/DC test method is same to that of the

black-box test method except the method name is changed to *testMethod_MDCC_Txx* (xx means the test data ID).

4. Test Execution
In this stage, just use customized JUnit test runner to run these MC/DC tests and the result together with some useful data are collected to prepare for the final analysis and comparison.

5.2.5 Result Collection and Comparison

The data collected from Black-Box testing and MC/DC testing on each tested COMWS method is listed in table 4.1.

Table 4.1 result data needed to be collected

Black-Box (BB) testing	MC/DC Testing
The number of BB tests	The number of MC/DC tests
The work (cost time) to write BB tests	The work (cost time) to write MC/DC tests
The number of faults found by BB	The number of faults found by MC/DC
Branch coverage of tested program	Branch coverage of tested program
Code coverage of tested program	Code coverage of tested program
	Extra number of MC/DC tests to find faults which BB testing didn't find

Three comparison criteria were used: Testing Cost, Faults Found and Program Coverage. Testing Cost (the number of tests and the time taken to write these tests) aims to find out whether MC/DC testing needed more tests and cost more time in writing the tests than Black-Box testing or not; Faults Found aims to find out whether MC/DC testing found extra faults that Black-Box testing didn't find; Program Coverage (Branch Coverage and Code Coverage) aims to find out whether MC/DC testing improved the coverage or not in testing the COMWS.

The comparison on these three criteria between Black-Box testing and MC/DC testing gives enough data in evaluating whether MC/DC is effective or not for testing the COMWS, so it makes a good preparation for final evaluation and conclusion.

5.3 Test Implementation

An example of MC/DC testing on the method `Login()` in the class Provider_Interface is shown below to indicate how this test implementation was carried out (the implementation of Black-Box testing on the method `Login()` is shown in Appendix A).

`Login()` is a fundamental method in COMWS that check whether user's input (email address and password) is correct to login the system. Its specification and

source code is shown in Figure 5.10 and 5.11, respectively.

```
/*Specfication of Login():
   Function:
      Login the system

   Program Inputs:
      Email:
      Originally we should consider 4 situations:
      1. length error email string (null, "" and too long string)
      2. wrong format email string (including symbols such as "'", ";" and so on)
      3. correct format but not existing email string
      4. correct format and existing email string

      But for situation 1:
      it will be checked in the front-end js file
      For situation 2:
      it will be checked by using the regular expression:
      "^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$" in the front-end js
      file
      so in this case, we only consider 3 and 4

      Password:
      Originally we should consider 4 situations:
      1. length error password string(null, "" and too long string)
      2. wrong format(Malicious) password string(including symbols such as ",",
      ";" to do the Database Injection)
      3. correct format but not 'correct' password string
      4. correct format and 'correct' password string

      But for sutuation 1:
      it will be checked in the front-end js file, so in this case, we only consider
      2,3 and 4

   Program Outputs:
      return value:
      "{'result': 0}" - wrong email or password
      "{'result': 1, 'id': .., 'username': ..}" - correct email and password
      (first .. means the id of the user, second .. means the user's name)
*/
```

Figure 5.10 Specification of Login()

```
public static String Login(String Email, String Password)
{
```

```
    String query = "SELECT * FROM provider WHERE Email = '"+Email+"'";
    ArrayList<Provider> plist = Provider.Search_Query(query);
    String jsondata = null;

    if(plist == null || plist.size() == 0){ // no such account
        jsondata = "{'result': 0}";
    }else{
        if(!Encode.authenticatePassword(plist.get(0).GetPassword(),
Password)){ //wrong password
            jsondata = "{'result': 0}";
        }else{ //correct email and password
            jsondata = "{'result': 1, 'id': " + plist.get(0).GetID() + ",
'username': '"+plist.get(0).GetUserName() +"'}";
        }
    }
    return jsondata;
}
```

Figure 5.11 Source Code of Login()

5.3.1 Black-Box Testing on Login()

The detailed implementation of Black-Box testing on the method `Login()` is shown in Appendix A.

5.3.2 MC/DC Testing on Login()

1. Boolean Expression Transformation

Based on the definition of 'equivalent complex Boolean expression' mentioned in section 2.3, the original Boolean expression shown in Figure 5.12 can be transformed into the complex Boolean expression shown in Figure 5.13.

```
if(plist == null || plist.size() == 0){
//no such account
}else{
if(!Encode.authenticatePassword(
            plist.get(0).GetPassword(),Password)){
//wrong password
}else{ //correct email and password }
}
```

Figure 5.12 original Boolean expression code in Login()

```
if(!(plist == null || plist.size() == 0) &&
        Encode.authenticatePassword(
              plist.get(0).GetPassword(),Password)){
              //focus on correct email and password
}else{//other situations}
```

Figure 5.13 equivalent complex Boolean expression

2. Test Cases Analysis and Tests Generation

Based on the MC/DC analysis template and equivalence complex Boolean expression for `Login()`, the detailed analysis process is shown in Figure 5.14. Besides the 3 same tests to Black-Box testing, one new test (T005) was generated to handle database error.

```
// Analysis:
//
// Conditions:
//  Line 37: plist == null
//  Line 37: plist.size() == 0
//  Line 43: (!Encode.authenticatePassword(plist.get(0).GetPassword(),
//           Password))
//
//   Based on the if-else relations, we can obtain a complex decision:
//   !(plist == null || plist.size() == 0) &&
//        Encode.authenticatePassword(plist.get(0).GetPassword(), Password)
//   it equals to:
//   plist != null && plist.size() != 0 &&
//        Encode.authenticatePassword(plist.get(0).GetPassword(), Password)
//
//   plist != null -> A
//   plist.size() != 0 -> B
//   Encode.authenticatePassword(plist.get(0).GetPassword(), Password) -> C
//
//   Test Cases:
//     CASE    A         B         C              Test
//     =====   ======================          =====
//     MDCC1   T         T         T              T002
//     MDCC2   T         T         F              T004
//     MDCC3   F         T         T*             T005
//     MDCC4   T         F         T*             T001
//
//  Note:
//     the * under C means don't care items
//
```

```
// [Test Data]
//         Test Cases          Inputs                Expected Outputs
//    ID    Covered   Email               Password     return value
//    T001  MDCC4     linanpp01@gmail.com 12345678     "{'result': 0}"
//    T002  MDCC1     linanpp09@gmail.com 123456
                        "{'result': 1, 'id': 1, 'username': 'Li Nan'}"
//    T004  MDCC2     linanpp09@gmail.com 1234567      "{'result': 0}"
//
//  For MDCC3, in order to make plist == null equals to true, I changed the correct
//  database DBPool into ErrorDBPool. By doing this, we can test whether plist
//  == null is true or not when using wrong database operation or information
//
//    T005  MDCC3     linanpp09@gmail.com 123456       "{'result': 0}"
//                    (Using error database)
```

Figure 5.14 MC/DC analysis and test generation for Login()

3. Test Realization

Based on the code template of MC/DC test method and the MC/DC test data for `Login()` generated in step 2, the detailed code for each MC/DC test of `Login()` method can be realized. Because the tests for test case MCDC1, MCDC2 and MCDC4 already exist in the Black-Box testing phase (T001, T002 and T004, see Appendix A), so there is no need to write these tests again. The test code (T005) for test case MCDC3 is shown in Figure 5.15. The detailed technique used in realizing this test is introduced in section 5.4.2.

```
@MyTest
public static boolean testLogin_MDCC_T005() {
    DBPool.closePool();
    ErrorDBPool.setupPool();
    DBSelect.select = 1;
    String result = Provider_Interface.Login("linanpp09@gmail.com", "123456");
    DBSelect.select = 0;
    ErrorDBPool.closePool();
    DBPool.setupPool();
    if(result.equals("{'result': 0}"))
        return true;
    else{
        reason=new String("Expected {'result': 0}, actual result :"+result);
        return false;}
}
```

Figure 5.15 new test to handle the database error

4. Tests Execution

Finally, just use my test runner to run these MC/DC tests (the same code to the tests execution phase of Black-Box testing). As the same to Black-Box testing, the console gives the execution result and the Eclemma gives the coverage result of this method, just shown in figure 5.16, 5.17 and 5.18.

```
java.sql.SQLException: Column 'UserName' not found.
        at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:910)
        at com.mysql.jdbc.ResultSet.findColumn(ResultSet.java:955)
        at com.mysql.jdbc.ResultSet.getString(ResultSet.java:5436)
        at org.apache.commons.dbcp.DelegatingResultSet.getString(DelegatingResultSet.java:263)
        at org.apache.commons.dbcp.DelegatingResultSet.getString(DelegatingResultSet.java:263)
        at Internship_Object.Provider.Search_Query(Provider.java:260)
        at Internship_Interface.Provider_Interface.Login(Provider_Interface.java:34)
        at Internship_Test.Provider_InterfaceTest.testLogin_MDCC_T005(Provider_InterfaceTest.java:367)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at Internship_Test.MyTestRunner.runTests(MyTestRunner.java:19)
        at Internship_Test.MyTestRunner.run(MyTestRunner.java:34)
        at Internship_Test.MyTestRunner.main(MyTestRunner.java:51)
Pool built succeed
Report
   runs=4
   fails=0
   successes=4
   executionFailures=0
```

Figure 5.16 MC/DC testing result on Login()

MyTestRunner (2013-11-5 0:22:23)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Login(String, String)	100.0 %	58	0	58

Figure 5.17 MC/DC testing code coverage on Login()

```
31    public static String Login(String Email, String Password)
32    {
33        String query = "SELECT * FROM provider WHERE Email = '"+Email+"'";
34        ArrayList<Provider> plist = Provider.Search_Query(query);
35        String jsondata = null;
36
37        if(plist == null || plist.size() == 0) // no such account
38        {
39            jsondata = "{'result': 0}";
40        }
41        else
42        {
43            if(!Encode.authenticatePassword(plist.get(0).GetPassword(), Password)) //wrong pass
44            {
45                jsondata = "{'result': 0}";
46            }
47            else //correct email and password
48            {
49                jsondata = "{'result': 1, 'id': " + plist.get(0).GetID() + ", 'username': '"+pl
50            }
51        }
52
53        return jsondata;
54    }
```

Figure 5.18 MC/DC testing branch coverage on Login()

Figure 5.16 indicates that all 4 MC/DC tests passed. The red exception hint means MC/DC actually finds out the database error. Figure 5.17 and 5.18 indicate its coverage result: 100% code coverage and 100% branch coverage, which means MC/DC can improve the branch coverage of `Login()` when comparing with Black-Box testing. These two points together with other data (for example: the number of MC/DC tests is 4 and extra tests number is 1) can be collected to prepare

for the final evaluation on `Login()` method.

The above testing on `Login()` is an example to show how to implement MC/DC testing on a single COMWS method. By using the same test procedure on other COMWS methods, useful result data can be collected to prepare for the final evaluation stage.

5.4 Test Problems

During the Black-Box testing and MC/DC testing on COMWS, there were two difficult points that need to be overcome:

(1) Because Servlet classes are used to communicate with mobile application, it is not easy to call these Servlet methods directly in Java application. So the way to test the Servlet side is the first difficult point.

(2) It is difficult to execute database error processing code if the server runs under correct database with correct commends. So testing these codes is the second difficult point.

This section focuses on how to use different techniques to solve these two problems.

5.4.1 Servlet Testing

Servlet in COMWS is acted as a bridge to connect between mobile application and back-end data processing functions. It receives the request from mobile application and returns back the result to complete the required task. But unfortunately, it is not easy to test Servlet directly in the Java application, which is due to two reasons.

(1) Servlet is not a simple class that its methods can be called directly in the Java application. It looks like a service that needs to be run in the Servlet container.

(2) Servlet can return back result only when it receives some requests from mobile application, but there is no mobile side to send different requests, that's the main reason why it is difficult to be tested.

Technique Solution:

Two steps were proposed to overcome the difficult listed above.

- **Using Tomcat to run Servlet**
  Tomcat is a Servlet container that can make Servlet works. So just put the COMP under the 'webapps' folder of the Tomcat Server and start Tomcat Server to run it. By doing this, these COMWS Servlet classes are activated and are available to response different requests at any time. This technique provides the fundamental requirement in testing Servlet of COMWS.

- **Write 'Client' class (acted as a mobile application) to send request**
  Because there is no mobile side application to send different requests to tested

Servlet, so writing 'Client' class to simulate mobile side's role is really necessary. The source code of 'Client' class is shown in Figure 5.19.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class Client {
   public static void sendGet(String url, String parameter){
    //no need to write}

   public static String sendPost(String url, String parameter){
        String result = ""; //store the servlet return result
        BufferedReader in = null;
        PrintWriter out = null;

        try{
            java.net.URL connURL = new java.net.URL(url); //connect url
            java.net.HttpURLConnection httpConn =
                (java.net.HttpURLConnection)connURL.openConnection();
            // set attribute
            httpConn.setRequestProperty("Accept", "*/*");
            httpConn.setRequestProperty("Connection", "Keep-Alive");
            httpConn.setRequestProperty("User-Agent", "Mozilla/4.0
                        (compatible; MSIE 8.0; Windows NT 6.1)");
            // Set post method
            httpConn.setDoInput(true);
            httpConn.setDoOutput(true);

            out = new PrintWriter(httpConn.getOutputStream());
            out.write(parameter); //send request parameter
            out.flush();
            //get the return back stream from servlet
            in = new BufferedReader(new
                InputStreamReader(httpConn.getInputStream(), "utf-8"));
            String line;
            //get the return back result
            while((line = in.readLine())!=null){
                result += line;
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally {
```

```
                try{
                    if(out!=null){
                        out.close();
                    }
                    if(in != null){
                        in.close();
                    }
                }catch(IOException ex){
                    ex.printStackTrace();
                }
            }
            return result; //return the return back result from servlet
        }
}
```

Figure 5.19 source code of Client class

As shown in Figure 5.19, the key method is `sendPost()` (because all the Servlets in COMWS handle post request, so there is no need to realize `sendGet()` method), which receives Servlet url and post parameter as parameters to simulate the process of sending post request to the tested Servlet. The return value of this method is the feedback from Servlet, which can be used to check whether the tested Servlet works properly or not.

By using such technique, testing Servlets of COMWS in Java application becomes available. For example, the LoginLet test template is shown in Figure 5.20. Just call the `sendPost()` method with 'link of LoginLet' and 'login tested data (post parameter)' and then compare the 'result' with the expected value to check whether LoginLet works properly or not.

```
@MyTest
public static boolean testLoginServlet_Txx(){
    String parameter = "{'email':'tested email', 'password':'tested password'}";
    String result =
       Client.sendPost(
       "http://localhost:8080/InternshipProject/Servlet/LoginLet",parameter);
    if(result.equals("expected value"))
       return true;
    else{
       reason = new String("Expected value, actual result:" + result);
       return false;
    }
}
```

Figure 5.20 LoginLet test template

5.4.2 Handle Database Error

Some methods in COMWS has *if*-condition code to handle database error, for example, as shown in Figure 5.21, for the method `ModifyProfile_Initial()`, the piece of code with red color is used to handle 'cannot get user name' database error.

```
public static String ModifyProfile_Initial(int UserID)
{
        String jsondata = null;

        if(UserID <=0)
        {
            jsondata = "{'err':'ID_error'}";
        }else{
               Provider p = new Provider(UserID);
               p.Search_ID();

               if(p.GetUserName() == null)
               {
                  jsondata = "{'err':'DB_error'}";
               }
               else if(p.GetUserName().equals(""))
               {
                  jsondata = "{'err':'ID_error'}";
               }
               else
               {
                  jsondata = "{'username':'"+p.GetUserName()+"',
                              'phonenumber':'"+p.GetPhoneNumber()+"',
                              'password':'"+p.GetPassword()+"',
                              'email':'"+p.GetEmail()+"'}";
                }
        }
        return jsondata;
}
```

Figure 5.21 red color code to handle database error

When using MC/DC to test this method, the analysis process is shown in Figure 5.22. The test case MDCC4 is related to the database error occur situation (`p.GetUserName()` actually equals to null), but unfortunately, it is difficult to write test for this test case by only consider the input parameter UserID because there is actually no wrong inside the database operation. So the result of such condition is the test for MDCC4 is hardly to be written and this piece of code is

unreachable.

```
// Analysis:
//  Conditions:
//      Line 76: UserID <= 0
//      Line 85: p.GetUserName() == null
//      Line 89: p.GetUserName().equals("")
//
//  Based on the if-else relations, we can obtain a complex decision:
//  !(UserID <= 0) && !(p.GetUserName() == null) && !(p.GetUserName().equals(""))
//
//  !(UserID <= 0) -> A
//  !(p.GetUserName() == null) -> B
//  !(p.GetUserName().equals("")) -> C
//
//    Test Cases:
//      CASE    A         B         C              Test
//      =====   ======================          =====
//      MDCC1   T         T         T              T002
//      MDCC2   T         T         F              T003
//      MDCC3   F         T         T              T001
//      MDCC4   T         F         T*             difficult!
//
//   Note:
//      the * under C means don't care items
```

Figure 5.22 MC/DC analysis on ModifyProfile_Initial()

Technique Solution:

Two steps were proposed to overcome this writing test difficult.

- **Create Error Database**

  In order to write tests for those database error-related test cases, using error database instead of original database to make error manually is a good way to satisfy it. So the first step is creating error database, which is shown in Figure 5.23.

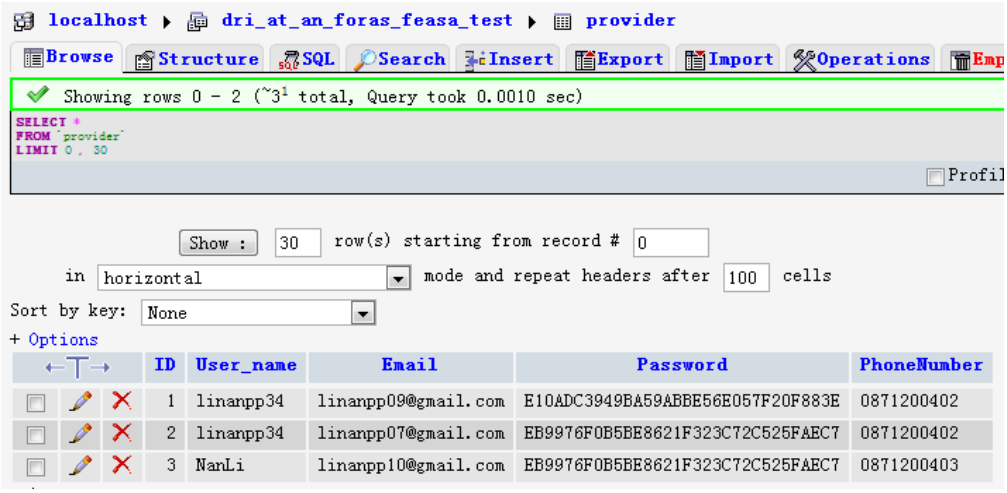


Figure 5.23 Error Database

The error database dri_at_an_foras_feasa_test is almost the same as original database except some table attributes are changed. For example, in order to realize test for MDCC4 mentioned above, the attribute 'Username' in the original database is changed to 'User_name' in the error database so that an error would be caught during the execution of `ModifyProfile_Initial()`, which makes '`p.GetUserName()`' equals to null and satisfy MDCC4 requirement.

- **Using parameter to select which database is under used**

The second step is writing code to control the selection of database. For example, when testing `ModifyProfile_Initial()` method, the tests for MDCC1, MDCC2 and MDCC3 should be executed under the correct database, while that for MDCC4 should be executed under the error database. In order to realize such selection, a static parameter called 'select' is used to perform the exchange between correct database and error database. The parameter code and one example are shown in Figure 5.24 and 5.25.

```
public class DBSelect {
public static int select = 0; // correct database
}
```

Figure 5.24 'select' parameter

```
public void Search_ID(){
    if(DBSelect.select == 0){
        DBPool db = null;
        try{
            db = new DBPool();
            String query = "SELECT * FROM provider WHERE ID = '"+ID+"'";
            ResultSet rs = db.execute_Result(query);
            if(rs.next()){
                this.SetID(rs.getInt("ID"));
```

```
                this.SetUserName(rs.getString("UserName"));
                this.SetEmail(rs.getString("Email"));
                this.SetPassword(rs.getString("Password"));
                this.SetPhoneNumber(rs.getString("PhoneNumber"));
                }
        }catch(Exception e){
            UserName = null;
            Email = null;
            Password = null;
            PhoneNumber = null;
            e.printStackTrace();
        }finally {
            try{
               if(db != null)
               db.close();
            }catch(Exception e){
               e.printStackTrace();
               }
        }
    }
    else // which is used for test
    {
        ErrorDBPool db = null;
        try
        {
            db = new ErrorDBPool();
            String query = "SELECT * FROM provider WHERE ID = '"+ID+"'";
            ResultSet rs = db.execute_Result(query);
            if(rs.next()){
                this.SetID(rs.getInt("ID"));
                this.SetUserName(rs.getString("UserName"));
                this.SetEmail(rs.getString("Email"));
                this.SetPassword(rs.getString("Password"));
                this.SetPhoneNumber(rs.getString("PhoneNumber"));
            }
        }catch(Exception e){
            UserName = null;
            Email = null;
            Password = null;
            PhoneNumber = null;
            e.printStackTrace();
        }finally {
            try{
                if(db != null)
```

```
                    db.close();
                }catch(Exception e){
                    e.printStackTrace();
                }
            }
        }
}
```

Figure 5.25 an example of using parameter 'select'

As shown in Figure 5.24, the rule for parameter 'select' is:
1. Equals to 0 -> select correct database
2. Equals to 1 -> select error database

Figure 5.25 indicates an example to show how this parameter is used. The method `search_ID()` is called inside `ModifyProfile_Initial()`, it directly decides whether the database error code will be executed or not. Based on the MC/DC analysis shown in Figure 5.22, the tests for MDCC1, MDCC2 and MDCC3 should be executed under the correct database, so the parameter 'select' should be equal to 0 so that method `search_ID()` will execute correct database (DBPool) -related code. On the contraries, the test for MDCC4 should be executed under the error database, so in such situation, the parameter 'select' should be equal to 1 so that method `search_ID()` will execute error database (ErrorDBPool) -related code.

By using parameter 'select' together with its related methods, it is available to write tests for those database error-related test cases. For example, the source code for `ModifyProfile_Initial()` MDCC4 is shown in Figure 5.26.

```
@MyTest
public static boolean testModifyInitial_MDCC4_T010() {
    DBPool.closePool();
    ErrorDBPool.setupPool();
    DBSelect.select = 1;
    String result = Provider_Interface.ModifyProfile_Initial(2);
    DBSelect.select = 0;
    ErrorDBPool.closePool();
    DBPool.setupPool();
    if(result.equals("{'err':'DB_error'}"))
        return true;
    else
    {
        reason=new String("Expected {'err':'DB_error'}, actual
result :"+result);
        return false;
```

```
    }
}
```

Figure 5.26 source code of ModifyProfile_Initial()

The procedure of this test code is:

(1) Close correct database pool (`DBPool.closePool()`) and start error database pool (`ErrorDBPool.setupPool()`)
(2) Using the 'select' parameter to select the error database (`DBSelect.select = 1`)
(3) Run the tested method (`ModifyProfile_Initial`) and obtain return result
(4) Recover to the original correct database set (`DBSelect.select = 0`; `ErrorDBPool.closePool()` and `DBPool.setupPool()`)
(5) Compare the return result with the expected return value to check whether the tested method (`ModifyProfile_Initial()`) executed properly or not.
(6) Finish the test.

5.4.3 Summary

The technique listed in section 5.4.1 and 5.4.2 provides a good environment for MC/DC to test COMWS. The 'Servlet Testing Technique' in section 5.4.1 makes it easy for MC/DC to test Servlet classes and the 'Handle Database Error Technique' in section 5.4.2 makes it possible for MC/DC to cover those database-error codes and improve the precision of testing results.

6. Test Results

This section provides the test results data collected from the test implementation stage. It shows the Black-Box testing and MC/DC testing data for each tested class and servlet and compares this data in three areas: Testing Cost, Faults Found and Program Coverage.

The tested classes and servlets are:

(1) `Provider_Interface` class: this class is used to provide user-related operation services (such as Login, Register, Modify Profile and so on) to the front-end website and mobile side users.
(2) `SymbolTransfer` class: this class is used to provide normal string-json string symbol transfer service to the back-end metadata storage method.
(3) `DeleteFile` class: this class is used to provide delete images service to the back-end web resource storage method.
(4) Servlets, includes:
    - LoginLet: it aims to response the 'login' request from mobile side and

returns login result

- RegisterLet: it aims to response the 'register a new account' request from mobile side and returns register result
- ViewMetaDataListLet: it aims to response the 'view current user's objects' request from mobile side and returns information of those cultural objects
- ViewSingleObjectLet: it aims to response the 'view info of a single cultural object' request from mobile side and returns information of the selected single cultural object

6.1 Class 'Provider_Interface'

Table 6.1 test results for class 'Provider_Interface'

Criteria		Methods			
		Login()	**ModifyProfile_Initial()**	**Register()**	**ModifyProfile()**
The Number of Tests	**BB**	4	9	3	12
	MC/DC	4	4	5	6
Time Costs (Minutes)	**BB**	63	60	57	86
	MC/DC	20	22	34	56
Faults Found	**BB**	0	0	0	1
	MC/DC	0	0	2	1(not the same to BB)
Branch Coverage	**BB**	83.3%	83.3%	75%	91.67%
	MC/DC	100%	100%	100%	100%
Code Coverage	**BB**	100%	94.6%	96.1%	96.0%
	MC/DC	100%	100%	100%	100%
Extra Test for MC/DC		1	1	3	2

Table 6.1 shows the extra cost and coverage for MC/DC, along with the extra faults found for the class `Provider_Interface`. Those results can be summarized into three factors: Testing Cost, Faults Found and Program Coverage. These are now considered in more detail.

☐ **Testing Cost**

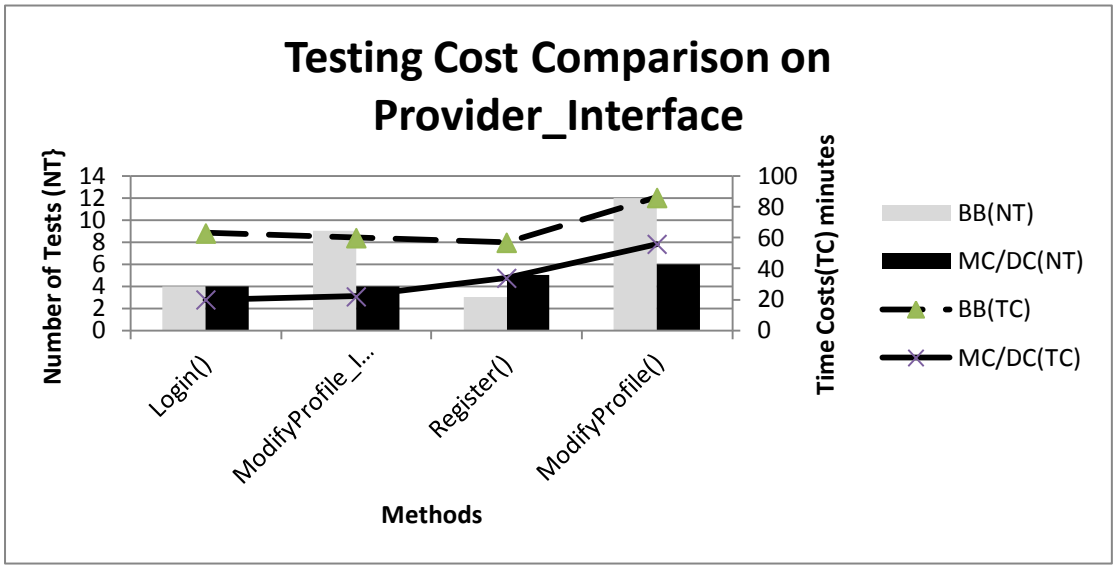


Figure 6.1 testing cost comparison on Provider_Interface

Figure 6.1 indicates the testing cost factor (the number of tests and time costs) results data for the class `Provider_Interface`. These results show two points:

1. For all 4 methods, developing Black-Box tests took significantly longer (almost twice longer time) than MC/DC tests.
2. For most `Provider_Interface` methods, the number of Black-Box tests was greater than (`ModifyProfile_Initial()` and `ModifyProfile()`) or equal to(`Login()`) that of MC/DC tests, but it is not the case for method `Register()`. This special method needed two more tests to complete MC/DC testing when compared with Black-Box testing.

☐ **Faults Found**

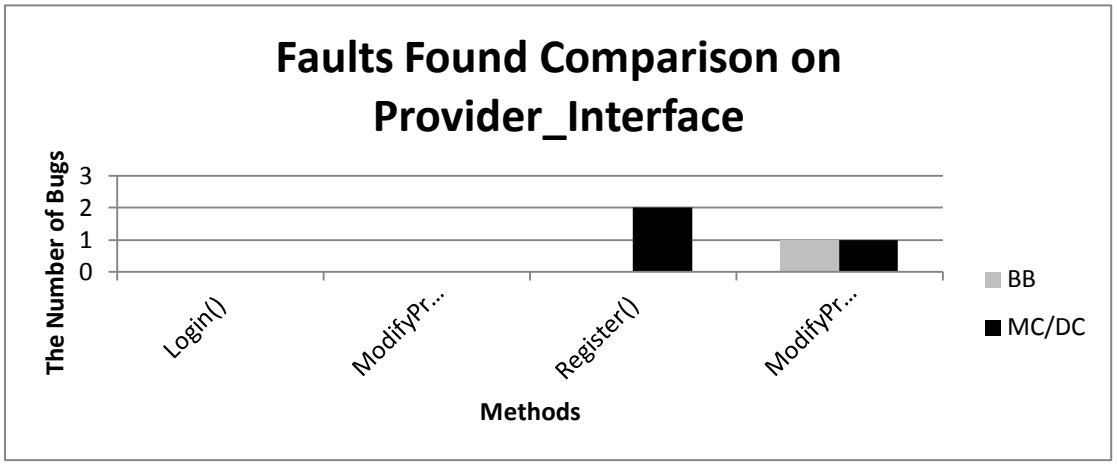


Figure 6.2 faults found vomparison on Provider_Interface

Figure 6.2 indicates the faults found factor results data for the class `Provider_Interface`. These results indicate two points:

1. For method `Login()` and `ModifyProfile_Initial()`, both the Black-Box testing and MC/DC testing did not find any errors.
2. For method `Register()` and `ModifyProfile()`, MC/DC testing found extra errors that Black-Box testing didn't find. `Register()` passed Black-Box testing while two errors were found by MC/DC testing. Although Black-Box testing found one error in `ModifyProfile()`, another one was still missed and detected by MC/DC testing.

☐ **Program Coverage**

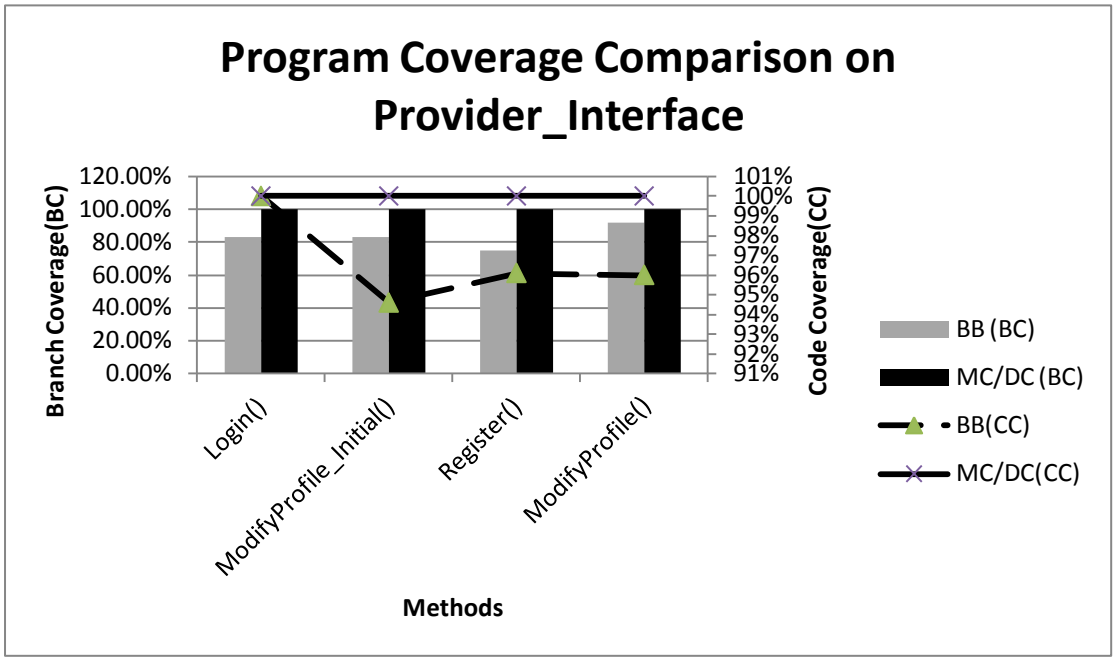


Figure 6.3 program coverage comparison on Provider_Interface

Figure 6.3 indicates the Program Coverage factor (code coverage and branch coverage) results data for the class `Provider_Interface`. These results indicate 2 points:

1. MC/DC testing reached 100% code coverage for all 4 methods. Although Black-Box testing reached 100% code coverage for method `Login()`, the rest three methods were not the case.
2. For all 4 methods, MC/DC testing also reached 100% branch coverage while only average 83.3% reached by Black-Box testing.

6.2 Class 'SymbolTransfer'

Methods in class `SymbolTtansfer`:
(1) `isLetter()`
(2) `isDigit()`
(3) `isHexLetter()`
(4) `isLetterOrDigit()`
(5) `isHexDigit()`
(6) `StringToJson()`
(7) `JsonToString()`

Table 6.2 test results for class 'SymbolTransfer'

Compare Criteria		Methods						
		(1)	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**(7)**
The Number of Tests	**BB**	15	9	15	21	21	9	22
	MC/DC	5	3	5	3	3		7
Time Cost (Minutes)	**BB**	80	45	90	120	120	96	125
	MC/DC	35	15	25	15	14		47
Faults Found	**BB**	0	0	0	0	0	0	2
	MC/DC	0	0	0	0	0		1
Branch Coverage	**BB**	100%	100%	100%	100%	100%	100%	75%
	MC/DC	100%	100%	100%	100%	100%		75%
Code Coverage	**BB**	100%	100%	100%	100%	100%	100%	94.8%
	MC/DC	100%	100%	100%	100%	100%		96.9%
Extra Test for MC/DC		0	0	0	0	0		1

Table 6.2 shows the extra cost and coverage for MC/DC, along with the extra faults found for the class `SymbolTtansfer`. Those results can be summarized into three factors: Testing Cost, Faults Found and Program Coverage. These are now considered in more detail.

☐ **Testing Cost**

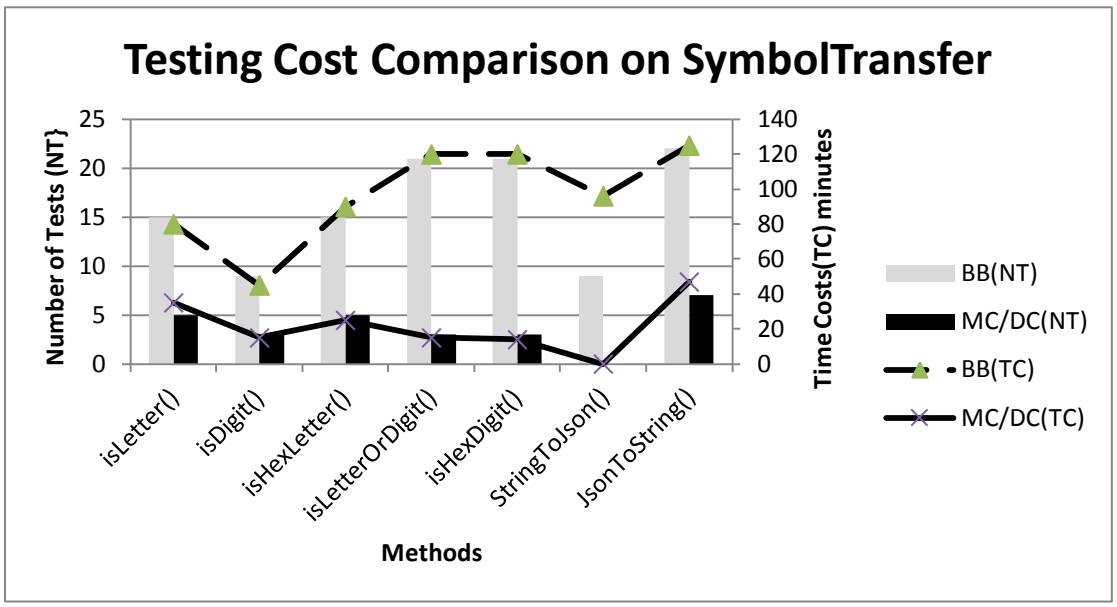


Figure 6.4 Testing Cost Comparison on SymbolTransfer

Figure 6.4 indicates the testing cost factor (the number of tests and time costs) results data for the class `SymbolTransfer`. These results show three points:

1. Method `StringToJson()` has no MC/DC tests.
2. for the other 6 methods:
   (1) Developing Black-Box tests took significantly longer than MC/DC tests.
   (2) The number of Black-Box tests was greater than MC/DC tests.
3. `isHexDigit()` had the maximum gap both in number of tests and time cost while `isDigit()` had the minimum gap both in these two areas.

☐ **Faults Found**

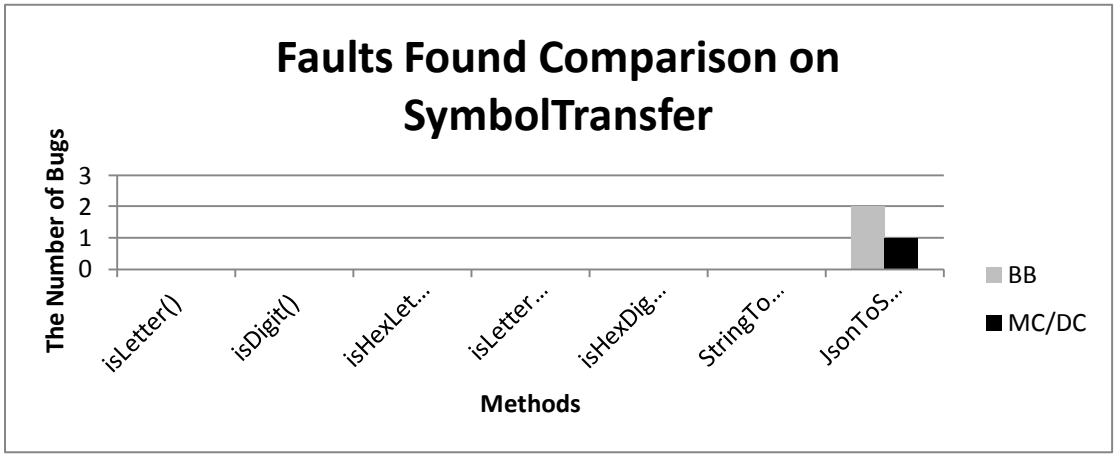


Figure 6.5 Faults Found Comparison on SymbolTransfer

Figure 6.5 indicates the faults found factor results data for the class `SymbolTransfer`. These results indicate two points:

1. For all methods except `JsonToString()`, both the Black-Box testing and MC/DC testing did not find any errors.
2. For method `JsonToString()`, MC/DC testing found extra errors that Black-Box testing didn't find. Although two errors were detected by Black-Box testing, one more was still missed and found by MC/DC testing.

☐ **Program Coverage**

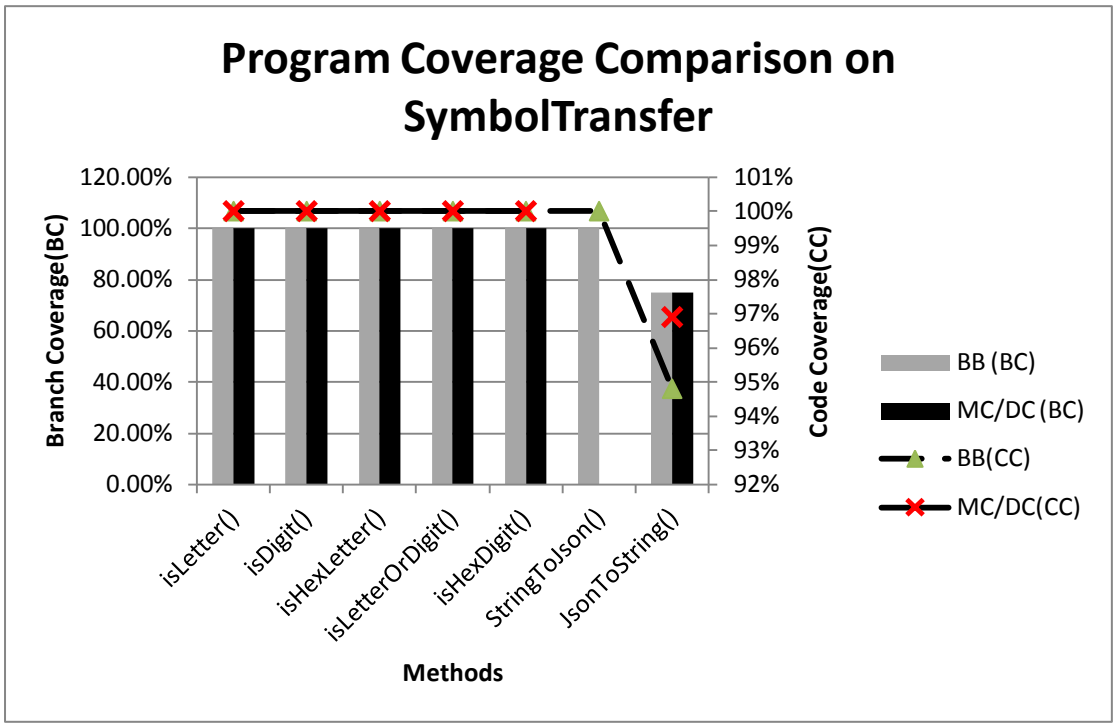


Figure 6.6 Program Coverage Comparison on SymbolTransfer

Figure 6.6 indicates the Program Coverage factor (code coverage and branch coverage) results data for the class `SymbolTransfer`. These results indicate three points:

1. `StringToJson()` reached 100% branch coverage and code coverage for Black-Box testing but had no coverage data for MC/DC testing.
2. For the other methods except `JsonToString()`, both Black-Box testing and MC/DC testing reached 100% branch coverage and code coverage.
3. `JsonToString()` had the same branch coverage (75%) for both Black-Box testing and MC/DC testing. Meanwhile, MC/DC testing (96.9%) reached a little bit higher code coverage than Black-Box testing (94.8%).

6.3 Class 'DeleteFile'

Table 6.3 test results for class 'DeleteFile()'

Compare Criteria		Methods	
		delFolder()	delAllFile()
The Number of Tests	BB	3	3
	MC/DC		6
Time Costs (Minutes)	BB	10	10
	MC/DC		45
Faults Found	BB	0	0
	MC/DC		2
Branch Coverage	BB	100%	71.4%
	MC/DC		100%
Code Coverage	BB	100%	60.7%
	MC/DC		100%
Extra Test for MC/DC			2

Table 6.3 shows the extra cost and coverage for MC/DC, along with the extra faults found for the class `DeleteFile`. Those results can be summarized into three factors: Testing Cost, Faults Found and Program Coverage. These are now considered in more detail.

☐ **Testing Cost**

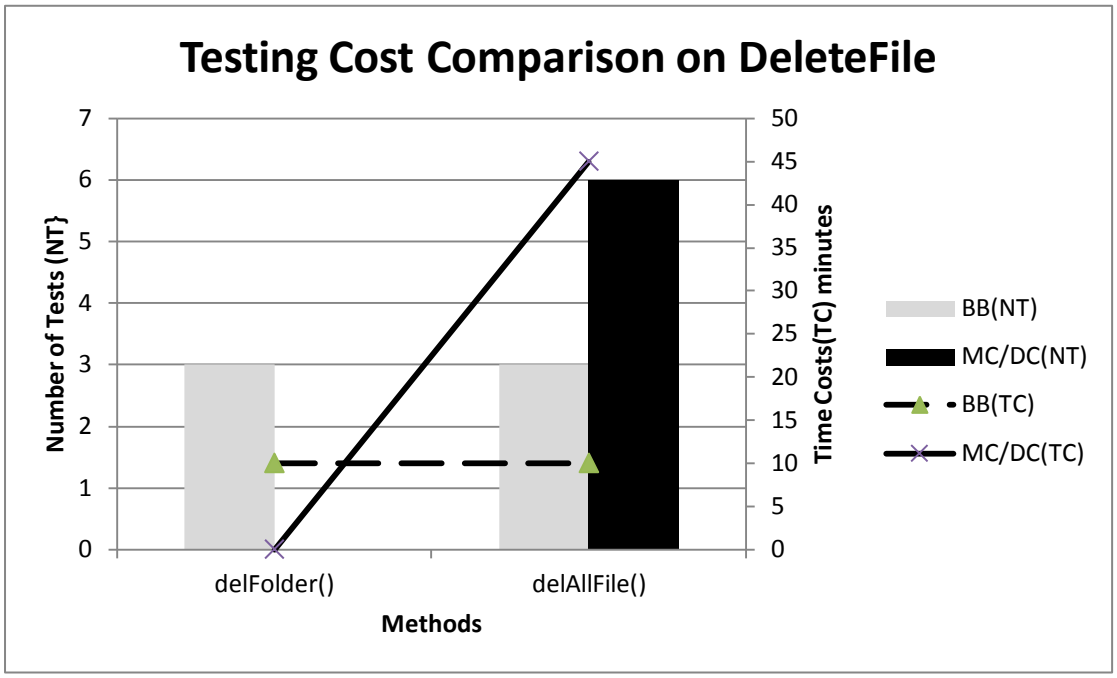


Figure 6.7 Testing Cost Comparison on DeleteFile

Figure 6.7 indicates the testing cost factor (the number of tests and time costs) results data for the class `DeleteFile`. These results show two points:

1. Method `delFolder()` has no MC/DC tests.
2. `delAllFile()` had more number of MC/DC tests than that of Black-Box tests and at the same time, needed twice time to develop MC/DC tests than Black-Box tests.

☐ **Faults Found**

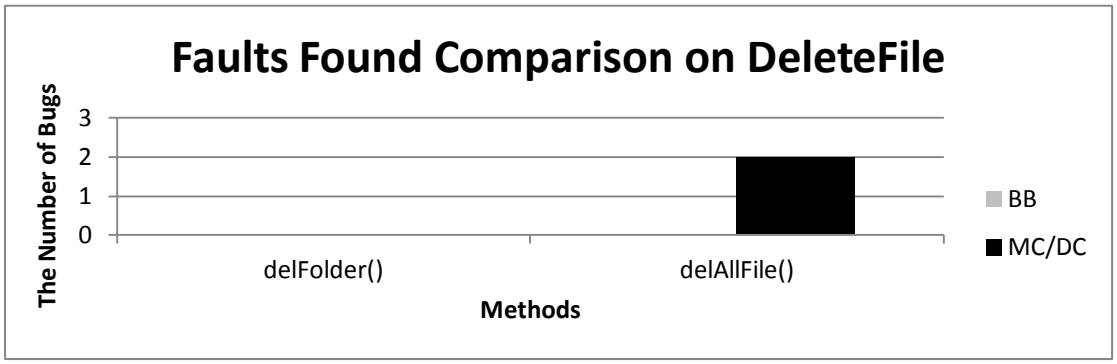


Figure 6.8 Faults Found Comparison on DeleteFile

Figure 6.8 indicates the faults found field results data for the class `DeleteFile`. These results indicate two points:

1. For method `delFolder()`, both the Black-Box testing and MC/DC testing did not find any errors.
2. For method `delAllFile()`, Black-Box testing found no error while MC/DC found two.

☐ **Program Coverage**

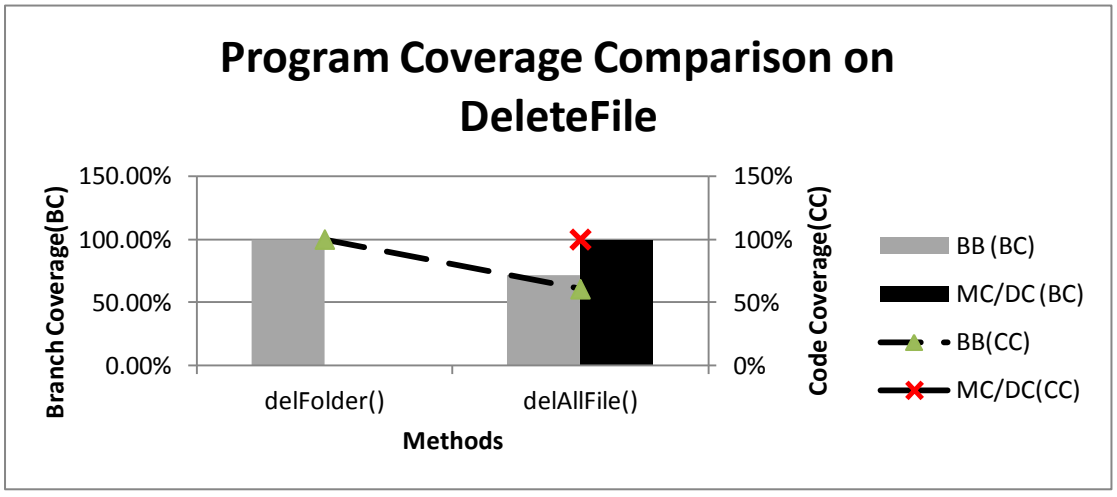


Figure 6.9 Program Coverage Comparison on DeleteFile

Figure 6.9 indicates the Program Coverage factor (code coverage and branch coverage) results data for the class `DeleteFile`. These results indicate two points:

1. `delFolder()` reached 100% branch coverage and code coverage for Black-Box testing but had no coverage data for MC/DC testing.
2. For method delAllFile(), MC/DC testing reached 100% branch coverage and code coverage while Black-Box testing just reached 71.4% and 60.7%, respectively.

6.4 Servlet Classes

Table 6.4 test results for servlet classes

Compare Criteria		Servlet				
		LoginLet	**RegisterLet**	**ViewMetaDataListLet**	**ViewSingleObjectLet**	**SingleImageUploadLet**
The Number of Tests	**BB**	5	6	5	5	9
	MC/DC	3	3	3	3	8
Time Costs (Minutes)	**BB**	40	48	40	40	63
	MC/DC	25	26	25	25	68
Faults Found	**BB**	0	0	0	0	0
	MC/DC	0	0	0	0	1
Extra Test for MC/DC		0	0	0	0	5

Table 6.4 shows the extra cost and coverage for MC/DC, along with the extra faults found for these Servlet classes. Those results can be summarized into three factors: Testing Cost, Faults Found and Program Coverage. These are now considered in more detail.

☐ **Testing Cost**

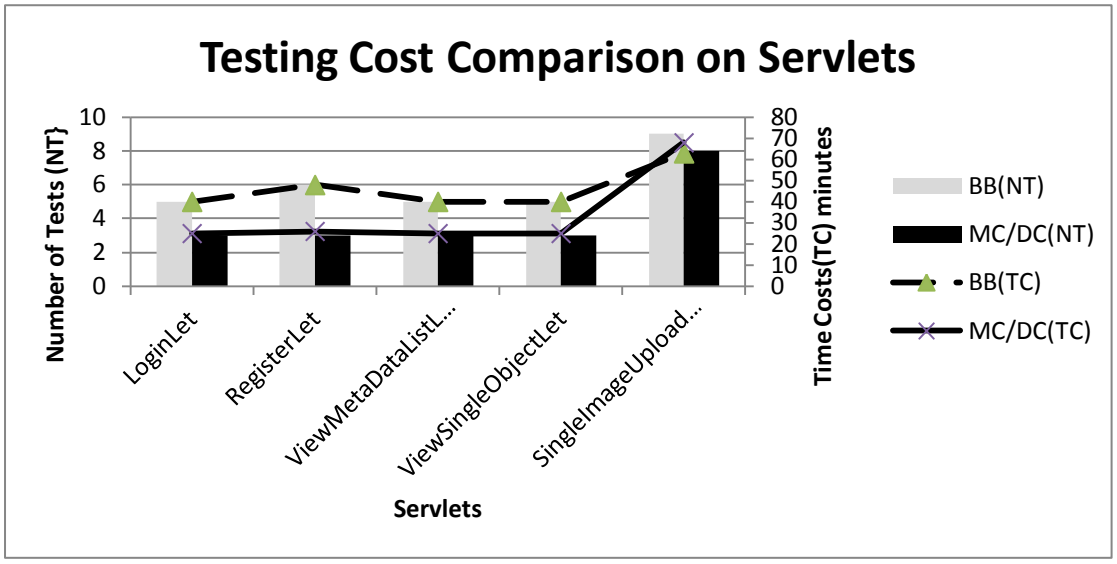


Figure 6.10 Testing Cost Comparison on Servlets

Figure 6.10 indicates the testing cost factor (the number of tests and time costs) results data for these Servlet classes. These results indicate two points:

1. For all Servlets, Black-Box testing needed more tests than MC/DC testing..
2. For almost all Servlets, developing Black-Box tests took significantly longer (almost twice longer time) than MC/DC tests. But it is not the case for `SingleImageUploadLet`, which needed a little bit longer time to develop MC/DC tests than Black-Box test.

☐ **Faults Found**

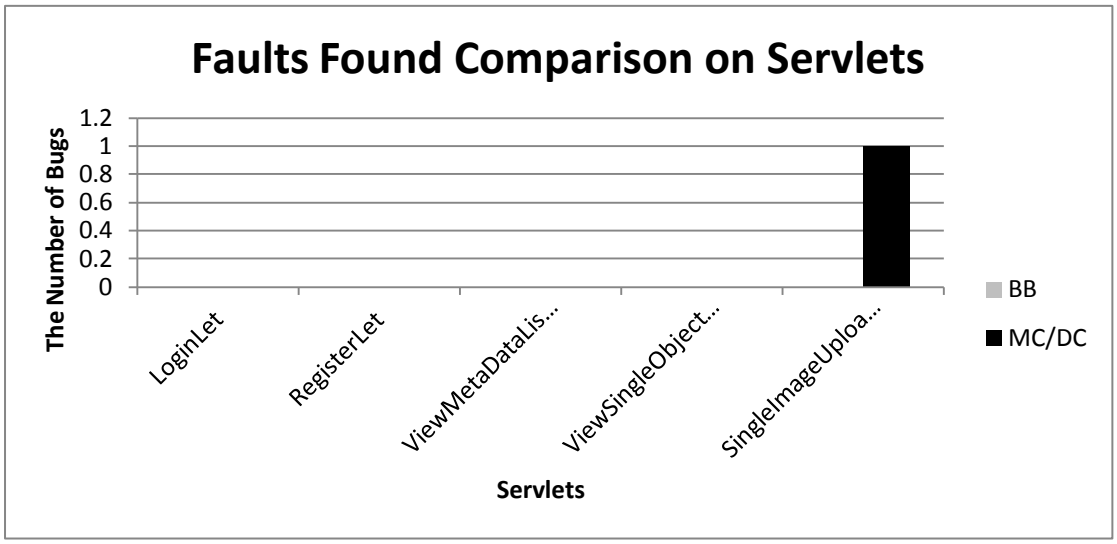


Figure 6.11 Faults Found Comparison on Servlets

Figure 6.11 indicates the faults found factor results data for these Servlet classes. These results show two points:

1. For all Servlets except `SingleImageUploadLet`, both Black-Box testing and MC/DC testing did not find any errors.
2. For Servlet `SingleImageUploadLet`, Black-Box testing found no error while MC/DC found one.

- **Program Coverage**

For Servlet classes, it is not easy to evaluate their code coverage and branch coverage by using Eclemma. The reason is shown in Section 7.3.3.

6.5 Results Summary

The results for each tested class method and Servlet listed above can be summarized into three points.

1. For almost all the tested methods and Servlets, developing Black-Box tests took significantly longer than MC/DC tests and at the same time, fewer MC/DC tests were needed than Black-Box tests.
2. MC/DC testing discovered some insidious but serious faults that Black-Box testing didn't find.
3. MC/DC testing improved code coverage and branch coverage of tested program.

In conclusion, MC/DC testing had a low testing cost while it found extra faults as well as providing improved code coverage and branch coverage of the tested program. Such results show that MC/DC was an additional effective testing technique to cooperate with Black-Box testing in testing COMWS.

7. Evaluation

This section evaluates the effectiveness of MC/DC in testing COMWS. The evaluation focuses on three factors: Testing Cost (Number of Tests and Time Costs), Faults Found and Program Coverage (Code Coverage and Branch Coverage). In each evaluation factor, it first discusses the reason why MC/DC was effective and could be acted as an additional testing technique to cooperate with Black-Box testing in testing COMWS and then, shows some exceptions which were different from these normal cases during testing.

7.1 Testing Cost

7.1.1 Number of Tests

Figure 7.1 summarizes the number of Black-Box tests and MC/DC tests for each tested method or Servlet.

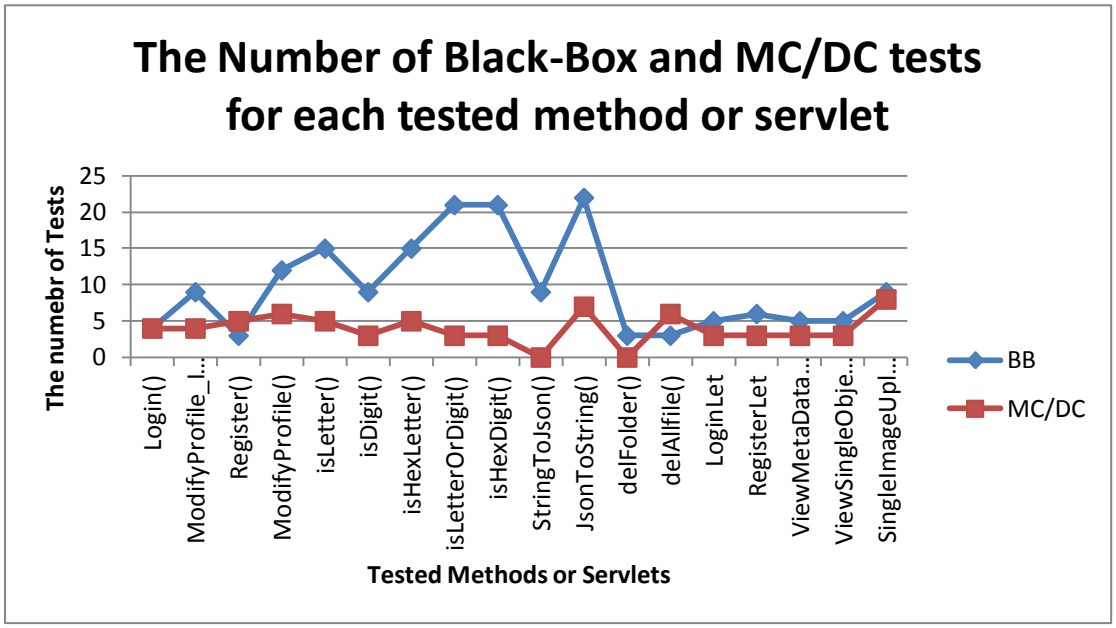


Figure 7.1 the number of tests for each tested method or servlet

As Figure 7.1 shows, for tested methods and Servlets, MC/DC required 57.8% of the number of Black-Box tests in average. There are three reasons:

(1) EP, BVA and CI were used to create Black-Box tests, each technique contributed some tests. Among these three techniques, typically BVA developed the most number of tests and was the immediate cause to enlarge the total number of Black-Box tests. There are two reasons:
- BVA selected the top and bottom value of each parameter's each partition as test case, so its tests were twice the number of EP tests.
- CI focused on discovering additional input combination tests that EP and BVA didn't cover, so the number of CI tests was unlikely to be very large.

(2) For a complex Boolean expression with n decisions, the number of MC/DC test cases (MC/DC tests) is n + 1. COMWS has a lot of complex Boolean expressions with 1, 2 or 3 decisions, and just few expressions have more than 5 decisions, which limit the number of MC/DC tests for the tested methods.

(3) MC/DC is an efficient testing technique. It only focuses on those complex Boolean expressions in the tested programs. Each MC/DC test case is related to

one specific situation of tested complex Boolean expression so that no duplicate tests are developed during MC/DC testing analysis. But this is not the case for Black-Box testing. For each partition of each parameter, EP selects its intermediate value as test while BVA selects its top and bottom value, so normally the test sets for these two techniques are different. But for `String` input parameter, because of its special equivalence partition, the test sets for EP and BVA may have intersection, which causes the situation of duplicate tests and makes Black-Box testing not efficient in testing software. The detailed analysis for this situation is shown in section 7.1.3.

However, two exceptions in this evaluation factor are shown in the Figure 7.1.

(1) MC/DC was not applicable for testing Methods `StringToJson()` and `delFolder()`. Take the method `delFolder()` for example.

```
public static void delFolder(String folderPath){
    try{
          delAllFile(folderPath);
          String filePath = folderPath;
          filePath = filePath.toString();
          File myFilePath = new File(filePath);
          myFilePath.delete();
     }catch(Exception e){
        e.printStackTrace();
     }
}
```

Figure 7.2 source code of delFolder()

Figure 7.2 shows the source code of `delFolder()`, which has no complex Boolean expressions. Because it contains no decision, MC/DC was not applicable for testing such method.

(2) Methods `Register()` and `delAllfile()` had an average 50% more MC/DC tests than Black-Box tests. Normally, the number of Black-Box tests is larger than that of MC/DC tests, but it was not the case for these methods. Take the method `delAllfile()` for example. Figure 7.3 shows the source code of `delAllFile()`.

```
public static boolean delAllFile(String path) {
      boolean flag = true;
      if(path == null)
         return false;

      File file = new File(path);
      if (!file.exists() || !file.isDirectory())
```

```
            return false;

        String[] tempList = file.list();
        File temp = null;
        for (int i = 0; i < tempList.length; i++) {
            if (path.endsWith(File.separator)) {
                temp = new File(path + tempList[i]);
            } else {
                temp = new File(path + File.separator + tempList[i]);
            }
            if (temp.isFile()) {
                temp.delete();
            }
            if (temp.isDirectory()) {
                delFolder(path + "/" + tempList[i]);
            }
        }
        return flag;
    }
```

Figure 7.3 source code of delAllFile()

From Black-Box testing view:

This method only has one input parameter `path`, which means there was no need to write CI tests (CI needs combination of parameters). At the same time, this input parameter only has three equivalence partitions (Null; correct folder path; incorrect folder path), which made this method only had 3 EP tests and 3 BVA tests (the same to EP). So the number of Black-Box tests for this method was limited.

From MC/DC testing view:

This method actually has three complex Boolean expressions:
1. `!file.exists() || !file.isDirectory()`
2. `path.endsWith(File.separator) && temp.isFile()`
3. `path.endsWith(File.separator) && temp.isDirectory()`

Each Boolean expression had 3 MC/DC test cases, although some test cases were used to jump to other Boolean expressions, this method still had 6 MC/DC tests.

In conclusion, for such methods which have a limited number of input parameters together with limited number of equivalence partitions for each input parameter, and at the same time, have quite a few complex Boolean expressions inside their realization, their number of MC/DC tests is larger than

that of Black-Box tests.

7.1.2 Time Cost

Figure 7.4 summarizes the cost time in developing Black-Box and MC/DC tests for each tested method and Servlet.

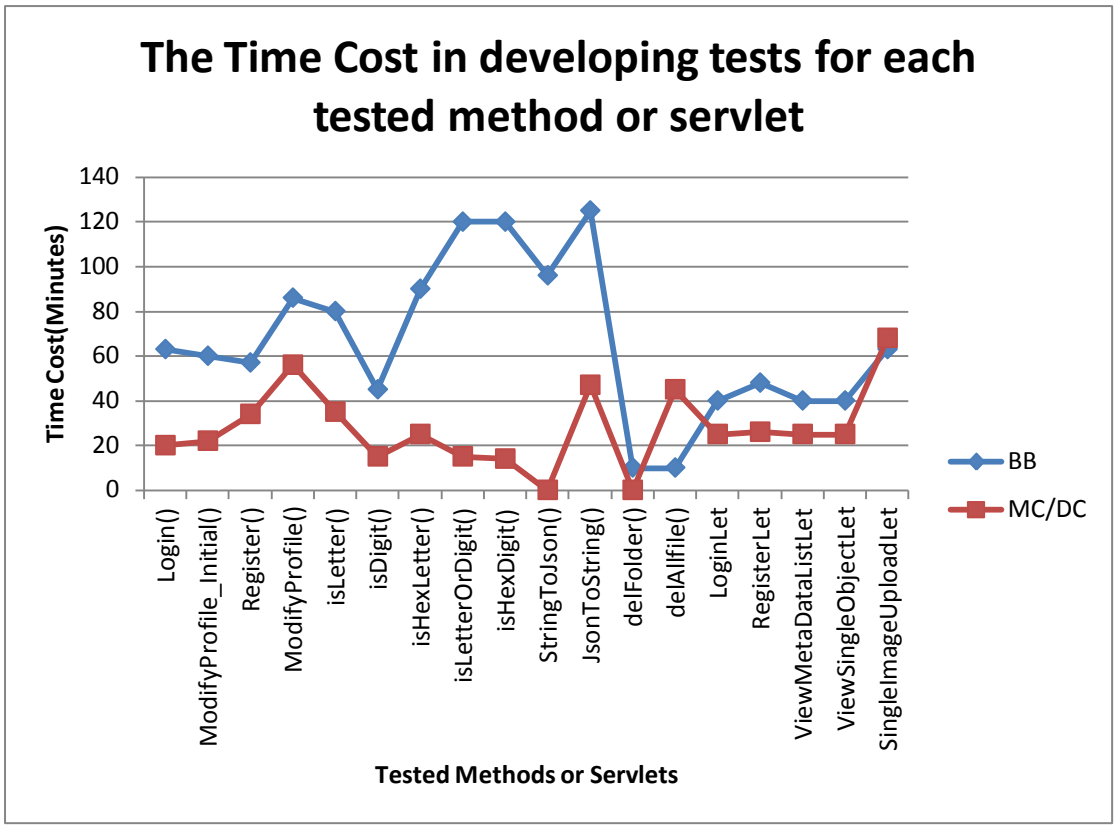


Figure 7.4 the time cost in developing BB and MC/DC tests for each tested method or servlet

As Figure 7.4 shows, for tested methods and Servlets, developing MC/DC tests took an average 64.41% time of developing Black-Box tests. There are two reasons:

(1) Three techniques were used to create Black-Box tests. Developing BVA and CI tests took up the most time in developing Black-Box tests.
BVA: this technique selected the bottom and the top values of each parameter's each partition as tests. Due to the large number of BVA test cases, BVA tests needed a lot of time to be generated.
CI: this technique considered every combination of input parameters, so the process of drawing truth table to analyze CI test cases and generating CI tests took a long time.

(2) As shown in 7.1.1, MC/DC is an efficient testing technique. It doesn't create duplicate tests and each test is related to one specific situation of tested complex Boolean expression. But for Black-Box testing, the tests set for EP and

BVA may similar, so it is a waste of time to write such duplicate tests, which makes Black-Box testing not very efficient.

However, two exceptions in this evaluation factor are shown in the Figure 7.4.

(1) MC/DC was not applicable for testing methods `StringToJson()` and `delFolder()`, so the time cost in developing MC/DC tests for these two methods was 0 minutes.

(2) Method `delAllfile()` and Servlet `SingleImageUploadLet` spent an average 42.57% more time in developing MC/DC tests than Black-Box tests.

Also take the method `delAllfile()` for example. There are two reasons:

- This method only has one input parameter, so there was no need to write CI tests. At the same time, because of its special equivalent partitions, the BVA tests were the same to EP tests. Based on these two points, developing Black-Box tests didn't spend too much time.

- But for MC/DC testing, because this method has three complex Boolean expressions, which made the process of analyzing MC/DC test cases became long, and at the same time, the test data for these test cases were hard to be generated (e.g. because of the condition code `file.isDirectory()`, MC/DC had to consider two situations: the folder path has no subfolder or has some subfolders. Such test data was no need to be considered in Black-Box testing. This special condition input is called 'Non-Parameter input', which is described in section 7.2), so these two points made developing MC/DC tests took extra time when compared with Black-Box tests.

7.1.3 Special Exception

By combining the number of tests and the time cost for each tested method or Servlet together, a special exception on method `Login()` and `Register()` was discovered.

Table 7.1 special exception

Compare Criteria		**Methods**	
		Login()	**Register()**
The Number of Tests	**BB**	**4**	3
	MC/DC	**4**	**5**
Time Costs (Minutes)	**BB**	**63**	**57**
	MC/DC	20	34

As shown in Table 7.1, for methods `Login()` and `Register()`, when compared with MC/DC testing, fewer Black-Box tests were developed but much more time

were needed. Such exception is due to the reason of duplicate tests for EP and BVA.

Take the method `Login()` for example. Based on the specification shown in Figure 5.4, `Login()` has two input parameters: `email` (string) and `password` (string). When considering the Black-Box testing analysis, because of the special equivalence partitions for `email` and `password`, the tests set for EP and BVA were actually 100% covered by each other (were the same). So the time for writing BVA tests was unnecessary.

It is a waste of time to write such duplicate tests, but this will never happen during MC/DC testing. MC/DC testing makes sure that its test cases are different from each other (because each MC/DC test case is related to the specific one truth table situation of the tested complex Boolean expression) and are combined together to meet MC/DC testing requirements. Such property makes MC/DC testing efficiently in testing software.

7.2 Faults Found

Figure 7.5 summarizes the number of faults found by Black-Box testing and MC/DC testing for each tested method and servlet.

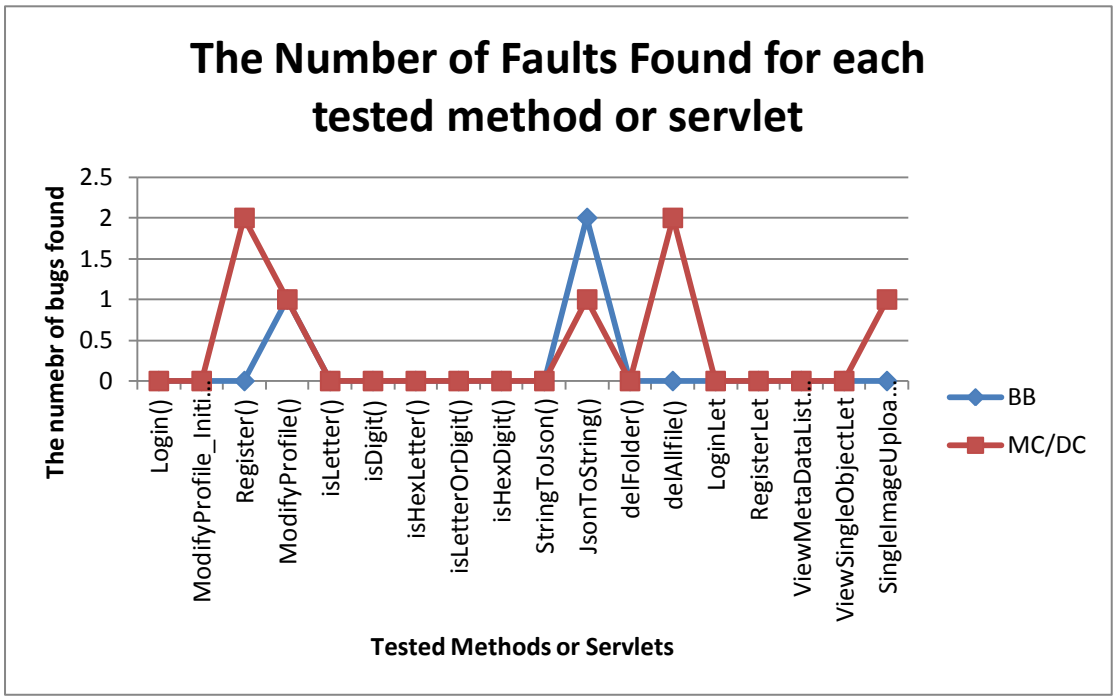


Figure 7.5 the number of faults found by Black-Box testing and MC/DC testing for each tested method or servlets

As Figure 7.5 shows, for 72.2% of the methods and servlets, both Black-Box and MC/DC testing didn't find faults. But for the remaining methods and servlets, MC/DC testing discovered an average of 1.4 extra faults that Black-Box didn't find.

The faults found by Black-Box testing were those small and obvious faults (e.g. miswritten fault, boundary value fault). One example is shown in Figure 7.6.

```
i = Integer.parseInt(t);
if ((i >= 0) && (i < 65536)) {//i<=65536 -> i<65536
    c = (char) i;
    curPos = tmpPos;
}
```

Figure 7.6 an error found by Black-Box testing

Figure 7.6 indicates a boundary value fault found by Black-Box testing. The value span of 'char' in JAVA is 0 to 65535. Originally the source code was `i<=65536`, but it was wrong because 65536 is not inside the correct range. By creating BVA tests and doing the Black-Box testing, such fault was found and fixed.

For MC/DC testing, the found faults were those insidious but serious faults that Black-Box testing did not find. There are two reasons:

(1) Black-Box testing focuses on the specification, and it doesn't not know the detailed implementation of each tested method, so it may miss some insidious faults.

(2) The 'Non-Parameter Input'. Some tested methods in COMWS are in the format of such code structure shown in Figure 7.7.

```
Method(input parameter1,2,3..) {
  ……
  if(which is not related to parameter value directly){ //non-parameter input
      ……
   }

  if(which is related to parameter value){
      ……
   }
  ……
}
```

Figure 7.7 the special 'Non-Parameter input' code format

Such methods contain code which is related to input parameters' value directly (**Parameter Input**), and at the same time, also include code that is not related to these parameters' value directly (**Non-Parameter Input**). The 'Parameter Input' code focuses most on the method's functionality realization and the 'Non-Parameter Input' code pays attention to the execution environment and

inner exception of the method. When testing such methods, MC/DC testing can find extra faults in these 'Non-Parameter Input' code while Black-Box testing cannot.

For Black-Box testing on such methods:
Because Black-Box testing doesn't care about the detailed implementation of tested method but just focus on the input parameters, so it actually focuses on these 'Parameter Input' code and try to find whether the tested method meets its specification or not.

For MC/DC testing on such methods:
MC/DC testing aims to check whether each condition in the decision in each tested method works properly or not. It focuses on the behavior of each condition and investigates the effect it has on the decision. So MC/DC testing actually focuses on both the 'Parameter Input' code and 'Non-Parameter Input' code so that can find insidious faults which exist in 'Non-Parameter Input' code.

For example, the code shown in Figure 7.8 is a 'Non-Parameter Input' fault that found by MC/DC testing.

```
public static boolean delAllFile(String path) {
   boolean flag = true;
   if(path == null)
        return false;

   File file = new File(path);
   if (!file.exists() || !file.isDirectory())
        return false;

   String[] tempList = file.list();
   File temp = null;
   for (int i = 0; i < tempList.length; i++) {
        if (path.endsWith(File.separator)) {
            temp = new File(path + tempList[i]);
        } else {
            temp = new File(path + File.separator + tempList[i]);
        }
        if (temp.isFile()) {
           temp.delete();
        }
        if (temp.isDirectory()) {
           // delAllFile(path + "/" + tempList[i]); //original code
           delFolder(path + "/" + tempList[i]); //new code
        }
```

```
        }
        return flag;
    }
```

Figure 7.8 a 'Non-Parameter Input' error found by MC/DC testing

Method `delAllFile()` aims to delete all files inside one folder. By using MC/DC testing analysis on this method, because of the existence of 'Non-Parameter Input' code '`temp.isDirectory()`', the situation about 'one folder that has subfolder' should also be taken into consideration. Based on the MC/DC testing result, the original code '`delAllFile(path + "/" + tempList[i]);`' under such decision was actually incorrect. That's because this code just deletes all files inside the subfolder while the subfolder itself is actually not deleted. The correct code should be '`delFolder(path + "/" + tempList[i]);`' , which can make sure that both the files inside subfolder together with subfolder itself are deleted successful.

But this 'Non-Parameter Input' fault could not be discovered by Black-Box testing. That's because the equivalence partitions for the input parameter path were just divided into three parts: null, correct folder path and incorrect folder path. This special 'Non-Parameter Input' situation 'one folder that has subfolder' was not considered in all three partitions.

7.3 Program Coverage

7.3.1 Code Coverage

Figure 7.9 summarizes the code coverage of Black-Box testing and MC/DC testing for each tested method.

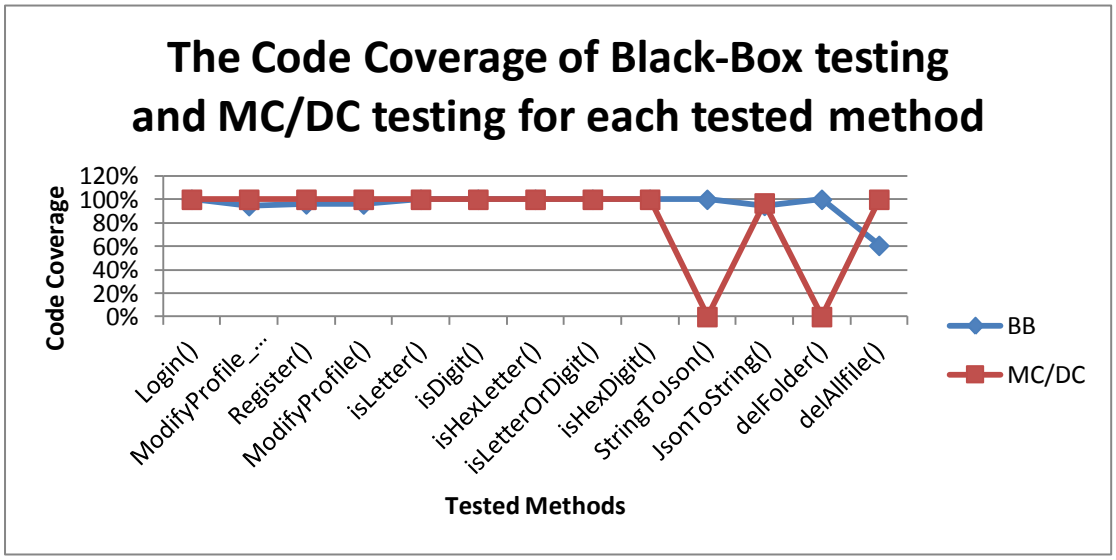


Figure 7.9 the code coverage of Black-Box testing and MC/DC testing for each tested method

As Figure 7.9 shows, for 33% of the methods, both Black-Box and MC/DC testing reached 100% code coverage. For the remaining methods, besides `StringToJson()` and `delFolder()`(MC/DC was not applicable for testing these two methods), MC/DC testing added an average 10.94% extra code coverage on the basis of Black-Box testing. The reason is:

Based on the 'Non-Parameter Input' mentioned in section 7.2, because Black-Box testing aims to check whether the tested program meets the specification or not, so it doesn't care the detailed implementation code inside the program. That's may make some 'Non-Parameter Input' code unexecuted during the testing process. On contrary, MC/DC testing focuses on the implementation code and detailed test cases are generated by analyzing those complex Boolean expressions inside the tested program, so these 'Non-Parameter Input' code is also under analyzed. MC/DC tests make sure that all situation-handle codes including 'Non-Parameter Input' code for the tested complex Boolean expression are executed during MC/DC testing.

Take the database error mentioned in section 5.3.2 for example. The code shown in Figure 7.10 was never covered during the Black-Box testing.

```
if(p.GetUserName() == null)
{
    jsondata = "{'err':'DB_error'}";
}
```

Figure 7.10 database error codes

Such code is actually the 'Non-Parameter Input' code in the tested method `ModifyProfile_Initial()`. This code has no relation with the input parameter `UserID`. So for the Black-Box testing, the tests never caused database error and the code '`jsondata = "{'err':'DB_error'}";`' was never executed during the testing. On contrary, MC/DC testing analyzed the code of tested program and one specific MC/DC test case focused on this 'Non-Paramter Input' code to check whether the tested program works properly if database error occurs. So the test implementation for this test case focused on replacing the correct database with an error-inside database to manually create database error, then checked whether the reaction of the tested program was correct or not. By doing this, the special 'Non-Parameter Input' code was executed, which made MC/DC testing added extra code coverage on the basis of Black-Box testing.

7.3.2 Branch Coverage

Figure 7.11 summarizes the branch coverage of Black-Box testing and MC/DC testing for each tested method.

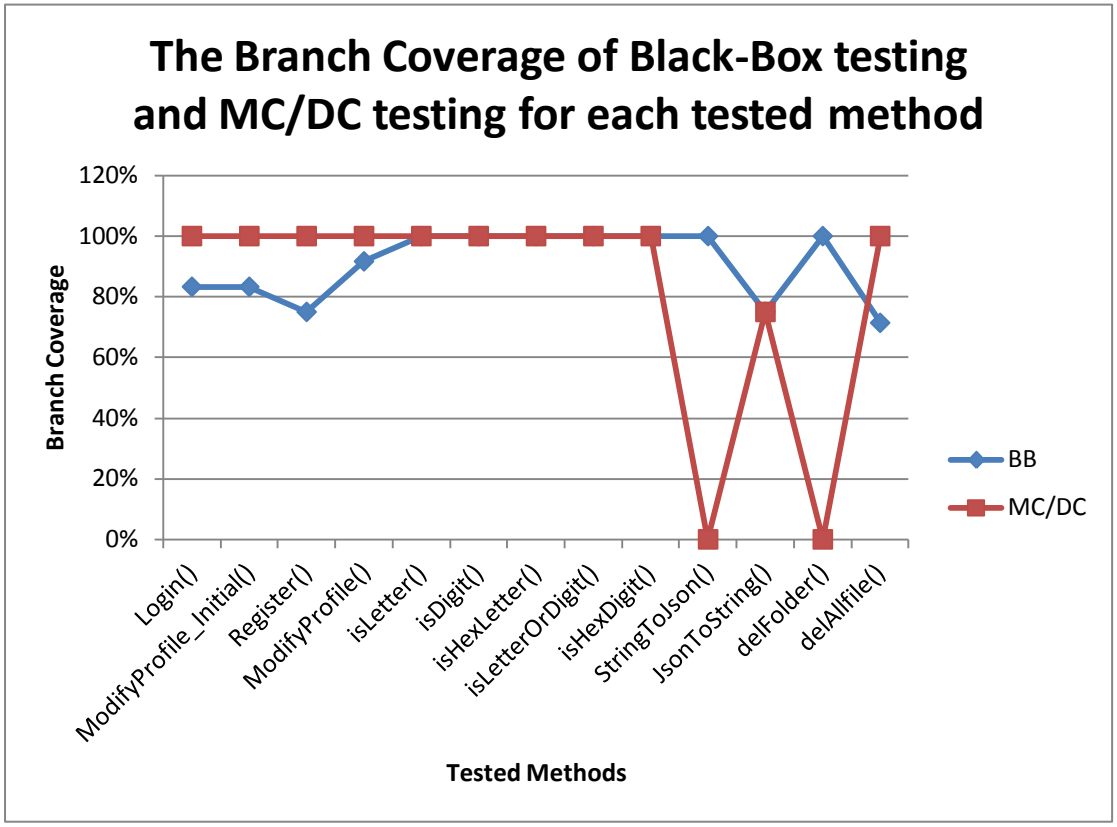


Figure 7.11 the branch coverage of Black-Box testing and MC/DC testing for each tested method

As Figure 7.11 shows, for 27.78% of the methods, both Black-Box and MC/DC testing reached 100% branch coverage. For the remaining methods, besides `StringToJson()` and `delFolder()`(MC/DC was not applicable for testing these two methods), MC/DC testing added an average 19.2% extra branch coverage on the basis of Black-Box testing. The reason is:

Based on the 'Non-Parameter Input' mentioned in 7.2, because Black-Box testing doesn't care the detailed implementation code inside the program, so some 'Non-Parameter Input' decision branches of complex Boolean expressions in the tested program may not be executed during the testing process. But this is not the case for MC/DC testing. MC/DC focuses on 'Non-Parameter Input' code inside the tested method, so its tests will make sure that every branch (T or F) of 'Non-Parameter Input' code is executed and covered at least once.

Take the `Login()` method for example. The code shown in Figure 7.12 checks whether the system has user's input account or not and Figure 7.13 shows the branch paths for this complex Boolean expression.

```
if(plist == null || plist.size() == 0) // no such account
    jsondata = "{'result': 0}";
else{…}
```

Figure 7.12 handle no such account codes in Login()

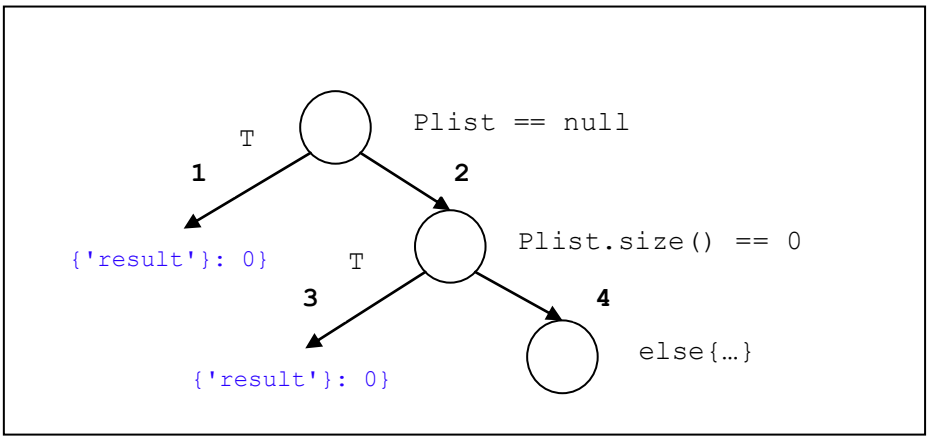


Figure 7.13 branch paths for this piece of Login() code

For Black-Box testing, those existing tests made branch 2,3 and 4 been covered during testing but there were no other tests to focus on testing database error to cover branch 1. That's because the database error code '`plist == null`' is actually the 'Non-Parameter Input' code of `Login()`, the input parameters `email` and `password` never make such database error occurs (Black-Box testing always used correct database during testing), so it is was impossible to let '`plist==null`' equals to true and branch 1 was never covered.

But for MC/DC testing, one specific test case was related to such 'Non-Parameter Input' database error to check whether `Login()` returns correct response string if database error occurs in the back-end side. So the test implementation for this test case replaced the correct database with an error-inside database to manually create database error, which made `plist==null` equals to true so that branch 1 was covered during the MC/DC testing process. By doing this, all branches including this special 'Non-Parameter Input' branch of this expression were covered, which made MC/DC testing added extra branch coverage on `Login()` on the basis of Black-Box testing.

7.3.3 Coverage for Servlets

It is easy to calculate code coverage and branch coverage for normal methods, but this is not the case for Servlets. Servlets don't run directly in the Java application, actually they are executed in the Servlet container (e.g. Tomcat). This situation makes it hard to calculate code coverage and branch coverage for Servlets (Because Eclemma is a Plug-in of MyEclipse and it works perfect for those programs that run directly in MyEclipse. One solution would be to use emma with the class files, but this was not explored in this project).

Although the evaluation technique was limited, the current Black-Box tests and MC/DC tests still reflect that MC/DC testing added extra code coverage and branch

coverage on the basis of Black-Box testing in testing those Servlets.

Take the Servlet `SingleImageUploadLet` for example. This servlet aims to receive uploaded single image from mobile side, store it into local server folder and return back upload result to mobile side. The most important part of code for this servlet that related to coverage criteria is shown in Figure 7.14.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
             throws ServletException, IOException {
       ...
       while(fii.hasNext())
       {
           ...
           if(!fis.isFormField() && fis.getName().length()>0)
           {
               ...
            }
        }
        ...
        if(!CultureObject_Interface.Modify_Upload_CultureObject_SingleWebReso
        urce(ObjectID, returnpath))
        {
           System.out.println("Database error"); //database error
        }
}
```

Figure 7.14 some part of code for singleimageuploadlet

Code Coverage:

Based on the 'Non-Parameter Input' mentioned in section 7.2, because the input parameters of this servlet has no relation to database error, so the Black-Box tests never caused database error and the line '`System.out.println("Database error")`' was never executed during the Black-Box testing on this servlet. But MC/DC testing analyzed and found this 'Non-Parameter Input' database error code, so a MC/DC test which replaced the correct database with error-inside database was developed to check whether the sentence 'Database error' was printed on the server console or not when the database error occurred. So this special MC/DC test made MC/DC testing added extra code coverage on the basis of Black-Box testing in testing this servlet.

Branch Coverage:

The branch paths for the complex Boolean expression '`!fis.isFormField() && fis.getName().length()>0`' is shown in Figure 7.15.

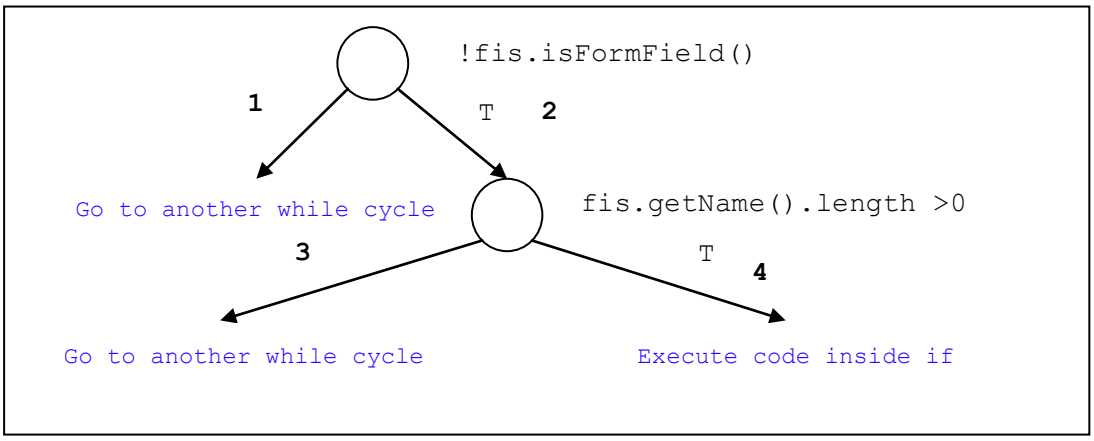


Figure 7.15 branch paths for one complex Boolean expression in singleimageuploadlet

`!fis.isFormField()` aims to check whether the uploaded data is text data or not. Because this servlet handles uploading image, so the first step is to make sure the uploaded data is not text data.

Based on the 'Non-Parameter Input' mentioned in section 7.2, for the Black-Box testing on this servlet, because the specification indicates that the input parameter (uploaded data) is image, so Black-Box tests never considered text data, which made the branch 1 never covered during testing. But for MC/DC testing, one specific MC/DC test focused on this 'Non-Parameter Input' situation (uploaded data is text data) to check whether the servlet works properly or not if text data is uploaded. So this special MC/DC test actually made MC/DC testing focused on 'Non-Parameter Input' code of tested servlet and added extra branch coverage on the basis of Black-Box testing in testing this servlet.

7.4 Summary

Based on the evaluation performed above, three key points can be summarized.

(1) MC/DC was an additional effective testing technique to cooperate with Black-Box testing in testing COMWS. It needed less testing cost while found extra faults that Black-Box testing didn't find as well as providing improved code coverage and branch coverage of tested programs.

- In testing cost factor, due to the reason of `String` input parameters of many tested methods and their special equivalence partitions, the test sets for EP and BVA were similar, which caused the existence of duplicate (inefficiency) tests. Writing such inefficiency tests was a waste of time and made Black-Box testing ineffective to test COMWS. This phenomenon was never happened in MC/DC testing. Each MC/DC test was related to one specific truth-table situation of considered complex decision, which means

that it was impossible to create duplicate MC/DC tests and thus MC/DC was effective in testing COMWS.

- In faults found factor, Because MC/DC testing focused on the behavior of each condition and investigated the effect it has on each decision in each tested method while Black-Box testing just focused on whether method meets with its specification, MC/DC testing found some insidious but serious faults that Black-Box testing did not find in testing COMWS.

- In program coverage factor, due to the reason of 'Non-Parameter Input', MC/DC testing added extra code coverage and branch coverage on the basis of Black-Box testing in testing COMWS.

(2) MC/DC showed its strength in testing programs with 'Non-Parameter Input' (as described in 7.2).

'Non-Parameter Input' is always used to check execution environment and handle inner exception of the tested method. Black-Box testing was not available to check faults inside 'Non-Parameter Input' code, that's because Black-Box tests were generated based on input parameters of the tested method, such tests actually didn't consider these 'Non-Parameter Input' code. But this is not the case for MC/DC. MC/DC testing focused on the complex Boolean expressions inside 'Non-Parameter' code, so there must be some MC/DC tests to focus on checking the correctness of 'Non-Parameter Input' code. That's the reason why MC/DC testing detected some insidious faults while Black-Box testing didn't.

(3) MC/DC also showed its advantage when compared with other white-box techniques in testing COMWS.

DCC: MC/DC had a stronger faults-finding ability than DCC in testing COMWS. Take `Register()` for example, Figure 7.16 indicates a complex Boolean expression that is used to return wrong format register error.

```
//code1 to check the format of username - if it is incorrect, then set ""
to username attribute of the user class object
//code2 to check the format of password - if it is incorrect, then set ""
to the password attribute of the user class object

if(p.GetUserName().equals("") || p.GetPassword().equals(""))
{
    jsondata = "{'result': -2}"; // wrong format
}
```

Figure 7.16 an example to show the strength of MC/DC over DCC

Originally, there was a fault in code 2 that it never sets empty string to the password attribute of the related user class object if password parameter has wrong format.

By using DCC, the test cases for this decision are:

1. {T, T}
2. {F, F}

Such test cases will hide this error because this decision will be true when the first condition is true and don't care what the value of second condition is. So if the test input is "wrong format username, wrong format password", the second condition is actually false but this method still returns correct result.

This is not the case for MC/DC. MC/DC test cases for this decision are:

1. {T, T}
2. {F, T}
3. {T, F}

The test input for test case 2 is "correct format username, wrong format password", which can be used to find this serious error.

MCC: MC/DC had a similar ability to find faults while only needed much less tests when compare with MCC in testing COMWS. For example, for the method `Register()` shown in Figure 2.7, MC/DC testing only developed 5 tests to find 2 errors and reach 100% code coverage and branch coverage while MCC needed 16 tests to achieve it.

Path Testing: Path testing causes every possible path from entry to exit of the tested program to be taken. But for some complex methods such as `JsonToString()`, it is difficult to test every path because the number of paths for those methods is incredible large. So MC/DC is a more available technique to test COMWS.

8. Conclusions

In this paper, an evaluation of the effectiveness of MC/DC in testing a specific web server (COMWS) was presented. This evaluation focused on three criteria: Testing Cost, Faults Found and Program Coverage to evaluate the effectiveness of MC/DC testing.

Based on the testing result and comparison evaluation, the essential conclusions are:

(1) MC/DC can be used as an additional effective testing technique to cooperate with Black-Box testing for testing web servers
(2) MC/DC is specially effective for 'Non-Parameter Input'
(3) Black-Box testing can be modified to be more efficient

8.1 Critical Analysis on Testing

(1) MC/DC can be used as an additional effective testing technique to enhance with Black-Box testing for testing web servers

When testing a web server, Black-Box testing is performed first to check whether these server methods meet their specifications or not. Such testing is the foundation of whole web server testing (Failed to pass Black-Box testing means didn't realize function, so there is no need to do other testing) and is used to find those small and obvious faults. But Black-Box testing has shortcomings. Black-Box tests are generated based on input parameters of the tested method, so such tests may not be available to detect faults inside 'Non-Parameter Input' code. In order to find these faults, another testing technique should be used to provide additional support. Based on the experiment result in testing COMWS together with detailed evaluation, MC/DC shows its high efficiency in testing web servers.

1. Based on the experiment result, when compared with Black-Box testing, MC/DC testing reduced the testing cost while found extra insidious but serious faults as well as improved code coverage and branch coverage of tested programs in testing COMWS. Such result is a reasonable evidence to indicate the effectiveness of MC/DC in testing web servers.

2. When compared with other white-box testing techniques, MC/DC also showed its strengths. For DCC, MC/DC had stronger ability to find faults than it; For MCC and Path testing, MC/DC showed higher feasibility than both of them.

(2) MC/DC is specially effective for 'Non-Parameter Input'

The inner code of some COMWS methods can be divided into two parts: 'Parameter Input' code and 'Non-Parameter Input' code. The first type code focuses on realizing the functionality of the tested method while the second one is used to provide method execution environment and handle inner exception of the tested method. Although the second type code has no relations with the value of input parameters, it still has effect on the tested method.

When performing Black-Box testing on COMWS, the tests were analyzed and generated based on the specification and input parameters of the tested method.

Typically specifications don't cover all the details of faults caused by lower level software. Such tests actually just focused on the 'Parameter Input' code and ignored 'Non-Parameter Input' code, which made Black-Box testing missed these special faults.

When performing MC/DC testing on COMWS, all complex Boolean expressions including expressions inside 'Non-Parameter Input' code were under analyzed. So there must be some MC/DC tests are used to check the correctness of 'Non-Parameter Input' code so that these insidious faults were succeeded to be detected.

Such 'Non-Parameter Input' is the main reason why MC/DC detected insidious faults that Black-Box testing didn't find.

(3) Black-Box testing can be modified to be more efficient

Section 7.1.3 shows the inefficient aspect in Black-Box testing. Normally, for each partition of each input parameter in the tested method, EP selects its medium value as test while BVA selects its top and bottom value, so the test sets for these two techniques are different, just as shown in Figure 8.1.

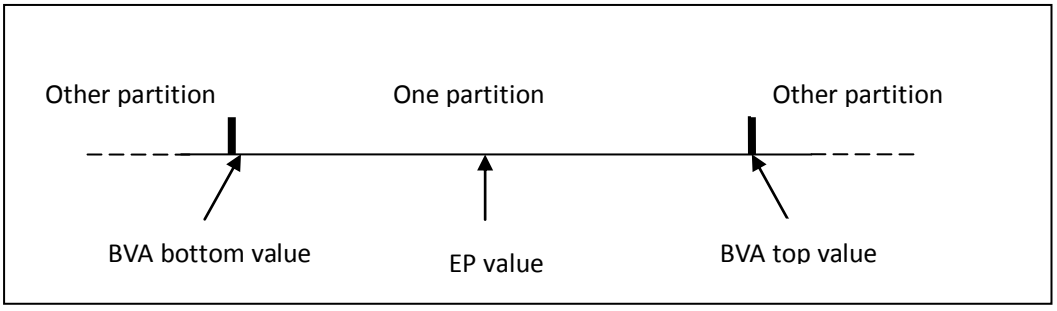


Figure 8.1 the test value selection for EP and BVA

But for `String` input parameter, because of its special equivalence partition, the test sets for EP and BVA may have intersection even totally covered by each other. For example, for the method `delAllFile()`,the EP and BVA testing analysis are shown in Figure 8.2.

```
// EP:
// [Test Cases] Partition {Test}
//   Inputs
//     path: [EP1] null
//           [EP2] correct folder path
//           [EP3] incorrect folder path
//   Outputs
//     return values: [EP4] false
//                    [EP5] true
```

```
//
// BVA:
// [Test Cases] Boundary Value {Test}
//   Inputs
//      path: [BVA1] null
//            [BVA2] correct folder path
//            [BVA3] incorrect folder path
//   Outputs
//      return values: [BVA4] false
//                     [BVA5] true
```

Figure 8.2 EP and BVA analysis for method delAllFile()

As shown in Figure 8.2, the test cases for EP and BVA are the same, which means the tests set for EP and BVA are 100% covered by each other. Because EP tests were developed first, so the development of BVA tests was a waste of time. Developing such duplicate tests make Black-Box testing not efficient in testing software. In order to improve the efficiency of Black-Box testing in testing software, a new test procedure is proposed.

(1) Based on the equivalence partitions of input parameters for the tested method, do the BVA testing first.

(2) Because there is no mandatory rule that EP should select the medium value of equivalence partition for each input parameter, so the top or bottom value of equivalence partition for each input parameter can be selected as EP tests value. Such EP tests already exist in the first step's BVA testing.

By doing this, there is no need to develop EP tests, so the time to develop Black-Box tests is reduced and the efficiency is improved. However, such technique also has disadvantage. Although it meets the requirement for EP and BVA, it only considers the boundary value of equivalence partitions for input parameters, so it doesn't look for faults at the middle of these equivalence partitions.

8.2 Limitations

Limitations found during the testing on COMWS are shown below:

(1) Coverage on Servlets

It is difficult to calculate code coverage and branch coverage for those tested Servlets. For normal methods, their code coverage and branch coverage can be calculated automatically by Eclemma, but this is not the case for Servlets. Servlets don't run directly in the Java application (they cannot be called through

their methods' name directly), actually they should be executed in the Servlet container (e.g. Tomcat). So such situation makes Eclemma hard to calculate coverage value for Servlets (Eclemma is a Plug-in of MyEclipse and it only works perfect for those pargrams that run directly inside MyEclipse).

(2) Shortcomings of MC/DC

While MC/DC ensures that a condition will not be masked out in a decision, it is still possible that the condition will ultimately be masked out within some sequence of statements in a program [WGY+13]. Figure 8.3 shows an example.

```
expr_1 = in_1 or in_2;   //stmt1
out_1 = expr_1 and in_3; //stmt2
```

Figure 8.3 an example to show the shortcoming of MC/DC

Based on the definition of MC/DC, one reasonable tests set (in the format {`in_1`, `in_2`, `in_3`}) for this program is:

1. {T, F, F}
2. {F, T, F}
3. {F, F, T}
4. {T, T, T}

The test cases with `in_3 = false` (bold faced) contribute towards MC/DC of `in_1` or `in_2` in stm1. Nevertheless, if the concentration is on the output variable `out_1`, the effect of `in_1` and `in_2` cannot be observed in the output since it will be masked out by `in_3 = false`. Thus, such test set gives MC/DC coverage of the program, but a fault on the first line will never propagate to the output (for example, if `in_1 or in_2` is miswritten into `in_1 and in_2`, such fault will never be found by the tests set listed above).

OMC/DC is designed to improve such shortcoming of MC/DC. It establishes observability of decisions by requiring that the variable whose assignment contains a particular Boolean decision remains unmasked through a path to a variable monitored by the test oracle [WGY+13]. By using OMC/DC, such mask out problem would be solved.

8.3 Critical Analysis of the Project

The current work of this project focused on evaluating the effectiveness of MC/DC in testing COMWS. Although a simple example shown in section 7.4 was used to indicate MC/DC has stronger ability to detect faults than DCC, there was no detailed experiment to compare the effectiveness of MC/DC and DCC in testing COMWS. At the same time, the effectiveness of MCC in testing COMWS should also be taken into

consideration so that the viewpoint 'MC/DC and MCC has similar ability to detect faults' can be experimented.

So if I did this project again, besides MC/DC testing and Black-Box testing, DCC and MCC testing should also be carried out in testing COMWS and other two comparisons between MC/DC and DCC, MC/DC and MCC were performed. These comparisons still focused on three criteria: Testing Cost, Faults Found and Program Coverage.

(1) Testing Cost: comparison was based on two areas: the number of tests together with the time taken to write these tests for MC/DC, DCC and MCC testing.

(2) Faults Found: comparison focused on the number of faults found by MC/DC, DCC and MCC testing. Particularly, found whether MC/DC detected faults that DCC didn't find and whether MCC detected faults that MC/DC didn't find.

(3) Program Coverage: comparison was based on two areas: the code coverage and the branch coverage of tested programs. Particularly, found whether MC/DC reached higher code coverage and branch coverage than DCC and whether MCC reached higher code coverage and branch coverage than MC/DC.

The evaluation on these three criteria can provide the evidence to show whether MC/DC actually performed better than DCC, and had the same faults-finding ability with MCC in testing web servers.

In addition, current work just focused on testing a small web server project which has limited number of methods, so it may hard to draw conclusions for larger projects.

8.4 Future Work

This section describes the future work that can be done to continue researching on MC/DC testing area.

- **Test more web server projects**

  Currently the MC/DC testing work just focused on the single web server COMWS and the experiment result were positive and encouraging. But one does not mean all, so such MC/DC testing and related evaluation should be carried out on more web server projects.

  For each tested web server project, the same comparison on three criteria (Testing Cost, Faults Found and Program Coverage) should also be carried out to evaluate the effectiveness of MC/DC in testing each web server. After the

evaluation for all tested web servers, it is necessary to combine those evaluation results together to verify whether MC/DC is actually effective in testing almost all web servers, not just a single one.

- **Test other programs**

  Based on the COMWS testing result and related evaluation, MC/DC is effective in testing web servers. So the question about 'whether MC/DC is still effective in testing other programs' is raised up. So MC/DC testing and related evaluation should also be carried out on other programs. Because web servers are stated-based systems, so for those non stated-based systems (e.g. normal Java programs), the performance of MC/DC in testing them is really worth to be researched.

  When start to do the MC/DC testing on other programs, the first step aims to calculate the numbers of complex Boolean expressions with n (1, 2, 3…) conditions inside the tested program. The detailed implementation for this process is shown in Appendix B. Then for the detailed testing part, the testing step is the same to that of testing COMWS and the result can be collected to evaluate whether MC/DC is still effective in testing those non state-based programs.

Reference

[LiH00] Liu KCH, Hsia P. (2000). *An object-oriented Web test model for testing Web applications*. In Proc. of IEEE 24th Annual International Computer Software and Applications Conference (COMP-SAC2000), Taipei, Taiwan, pp 537-542.

[ChM94] Chilenski JJ, Miller SP. (1994). *Applicability of modified condition/decision coverage to software testing*. Software Engineering Journal, Vol.7, No.5, pp 193-200.

[Whi01] White AL. (2001). *Comments on Modified Condition/Decision Coverage for software testing*. In 2001 IEEE Aerospace Conference Proceedings, 10-17 March 2001, Big Sky, Montana, USA, vol 6, pp 2821-2828.

[HaV01] Hayhurst KJ, Veerhusen DS. (2001). *A practical approach to modified condition / decision coverage*. In 20th digital avionics systems conference (DASC), Daytona Beach, Florida, USA, 14-18 October 2001, vol 1, pp 1B2/1-1B2/10.

[RTC92] RTCA, Inc. (1992). *RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Cerification*. Washington, D.C.

[USF93] U.S.Department of Transportation, Federal Aviation Administration. (1993). *Advisory Circular #20-115B.*

[Gal04] Galin D. (2004). *Software Quality Assurance: From Theory to Implementation.* Addison-Wesley.

[BTL+11] Brown S, Timoney J, Lysaght T, Ye DS. (2011). *Software Testing: Principles and Practice*. First Edition, China Machine Press, Beijing, China.

[Pre10] Pressman RS. (2010). *Software Engineering: A Practitioner's Approach*. Seventh Edition, McGraw-Hill, New York, NY.

[HVC+01] Hayhurst KJ, Veerhusen DS, Chilenski JJ, Rierson LK. (2001). *A parctical tutorial on modified condition/decision coverage*. NASA Technical Memorandum TM-2001-210876, NASA Langley Research Center, Hampton, VA.

[Chi01] Chilenski JJ. (2001). *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. FAA Tech Center Report DOT/FAA/AR-01/18.

[KaB03] Kapoor K, Bowen JP. (2003). *Experimental evaluation of the variation in effectiveness for DC, FPC and MC/DC test criteria*. In proceedings of ACM-IEEE 2003 international symposium on empirical software engineering (ISEIS 2003), Rome, Italy, 30 September-1 October 2003. IEEE Computer Society Press, pp 185-194.

[JoH01] Jones JA, Harrold MJ. (2001). *Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage.* In ICSM 2001, Proceedings of the International Conference on Software Maintenance, Florence, Italy, November 6-10, 2001, pp 92-101.

[PZL+05] Pan L, Zou B, Li JY, Chen H. (2005). *Bi-Objective Model for Test-Suite Reduction based on Modified Condition/Decision Coverage*. Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing, pp. 235-244.

[PMG+10] Prabhu J, Malmurugan N, Gunasekaran G, Gowtham R. (2010).*Study of ERP Test-Suite Reduction Based on Modified Conditon/Decision Coverage*. In Proceedings of the second international conference on computer research and development, Kuala Lumpur, 7-10 May 2010, pp 373-378.

[WGY+13] Whalen M, Gay G, You D, Heimdahl M, Staats M. (2013). *Observable Modified Condition/Decision Coverage*. In Proceedings of the 2013 International Conference on Software Engineering(ICSE 2013), San Francisco, CA, USA, pp 102-111.

[ViB06] Vilkomir SA, Bowen JP. (2006). *From MC/DC to RC/DC: formalization and analysis of control-flow testing criteria*. Formal Aspects of Computing, vol.18, no.1, pp 42-62.

[DuL00] Dupuy A, Leveson N. (2000). *An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software*. Proceedings of the Digital Aviation Systems Conference (DASC), Philadelphia.

[Bis03] Bishop PG. (2003). *MC/DC based estimation and detection of residual faults in PLC logic networks*. In Supplementary proceedings 14th international symposium on software reliability engineering (ISSRE '03), Fast Abstracts, Denver, Colorado, USA, 17-20 November, pp 297-298.

[CBE+04] Chopra V, Bakore A, Eaves J, Galbraith B, Li S, Wiggers C. (2004). *Professional Apache Tomcat 5*. John Wiley & Sons.

[MyS01] MySQL AB. *MySQL*. (2001). [Online]. Available: http://www.mysql.com/.

[Hof11a] Hoffmann MR. (2011). *Eclemma - jacoco java code coverage library*. [Online]. Available: http://www.eclemma.org/jacoco/index.html.

[Hof11b] Hoffmann MR. (2011). *Eclemma - java code coverage for eclipse*. [Online]. Available: http://www.eclemma.org/.

[DGH+10] Doerr M, Gradmann S, Hennicke S, Isaac A, Meghini C, vandeSompel, H. (2010). *The Europeana Data Model (EDM)*. In World Library and Information Congress: 76th IFLA General Conference and Assembly, 10-15 August 2010, Gothenberg, Sweden.

[Dub03] Dublin Core Metadata Initiative. (2003). *Dublin Core Metadata Element Set, Version 1.1. Reference Description*, retrieved October 31, 2005 from http://dublincore.org/documents/dces.

Appendix A. Black-Box Testing Design and Implementation Example

☐ Black-Box Testing Design

In black-box testing stage, for each tested method, a three steps procedure is required to check whether it meets specification requirement or not. Such procedure is shown in Figure A.1.

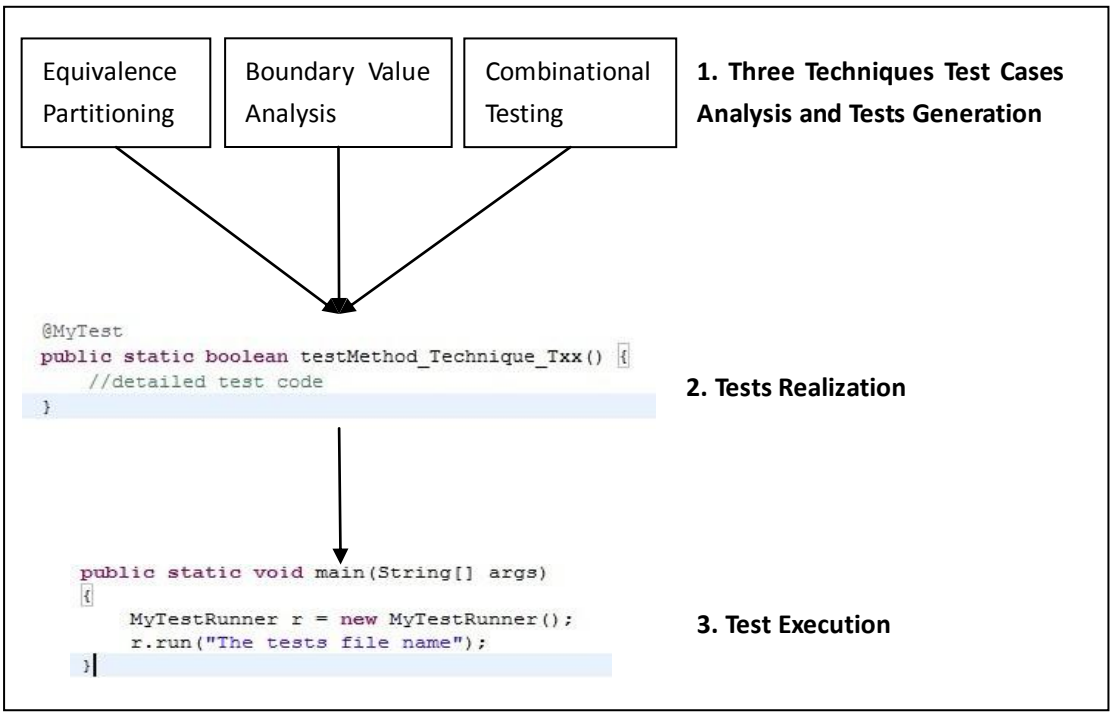


Figure A.1 3 steps procedure for Black-Box testing

1. Three Techniques Test Cases Analysis and Tests Generation

Three techniques are used to do black-box testing on COMWS: Equivalence Partitioning (EP), Boundary Value Analysis (BVA) and Combinations of Inputs (CI). Each technique needs different test analysis.

☐ **EP**

EP [BTL+11] is based on selecting representative values of each parameter from the equivalence partitions. This technique aims to check whether the basic data processing aspects of the COMWS methods are correct or not. The analysis template for EP is shown in Figure A.2.

```
//Analysis:
// [Test Cases] Partition {Test}
```

```
//    Inputs
//
//      Parameter 1: [EP1]...
//                   [EP2]...
//                   ...
//      Parameter 2: [EPk]...
//                   [EPk+1]...
//                   ...
//      ...
//    Outputs
//      return values: [EPn]...
//                     [EPn+1]...
//                     ...
```

Figure A.2 analysis template for EP

As Figure A.2 indicates, both the inputs and the outputs of the tested COMWS methods should be considered and each equivalence partition for each of the parameters is a test case. Based on these EP test cases, combined with the tested method's inputs, the EP test data of this method can be produced, just as shown in Figure A.3.

```
//  Test Data:
//        Test Cases          Inputs              Expected Outputs
//    ID   Covered        xxx    xxx   xxx  ...   return value
//   Test1 EP1,EPk,      ...    ...   ...  ...   ...
//         ..,EPn
//   Test2 EP2,EPk+1,    ...    ...   ...  ...   ...
//         ..EPn+1
//   ...   ...           ...    ...   ...  ...   ...
```

Figure A.3 EP test data generation

- **BVA**

BVA [BTL+11] focuses on the boundary conditions of each partition (the bottom and the top values) to check whether the COMWS methods have boundary-related programming faults. The analysis template for BVA is shown in Figure A.4.

```
//Analysis:
// [Test Cases] Boundary Value {Test}
//    Inputs
//      Parameter 1: [BVA1]...
//                   [BVA2]...
//                   ...
```

```
//        (related to the bottom and top values of EP of parameter 1)
//      Parameter 2: [BVAk]...
//                   [BVAk+1]..
//                   ...
//        (related to the bottom and top values of EP of parameter 2)
//      ....
//   Outputs
//      return values: [BVAn]...
//                     [BVAn+1]...
//                      ...
//        (related to the bottom and top values of EP of return values)
```

Figure A.4 analysis template for BVA

As Figure A.4 indicates, both the inputs and the outputs of the tested COMWS methods should be considered and each boundary value of equivalence partition for each of the parameters is a test case. Based on these BVA test cases, combined with the tested method's inputs, the BVA test data of this method can be produced, just as shown in Figure A.5.

```
//  Test Data:
//        Test Cases        Inputs            Expected Outputs
//    ID   Covered      xxx    xxx   xxx  ...   return value
//   Test1 BVA1,BVAk,   ...    ...   ...  ...   ...
//         ..,BVAn
//   Test2 BVA2,BVAk+1, ...    ...   ...  ...   ...
//         ..BVAn+1
//   ...   ...          ...    ...   ...  ...   ...
```

Figure A.5 BVA test data generation

- **CI**

CI [BTL+11] focuses on testing all the possible combinations of inputs of each COMWS tested method. It provides additional coverage that EP and BVA may not cover. The analysis template for CI is shown in Figure A.6.

```
//Analysis:
// Causes:
// 1. ...
// 2. ...
// 3. ...
// ...
// Effects:
// 1. ...
// 2. ...
// ...
```

```
//
// [Test Cases] Truth Table {Test}
//                                          Rules
//                                    1    2    3    4   ...
//    Causes
//    ....(for 1)                     xx   xx   xx   xx  ...
//    ....(for 2)                     xx   xx   xx   xx  ...
//    ...                              ..   ..   ..   ..  ...
//    Effects
//    ....(for 1)                     xx   xx   xx   xx  ...
//    ....(for 2)                     xx   xx   xx   xx  ...
//    ....                             ..   ..   ..   ..  ...
//                               Test   CI1  CI2  CI3  CI4 ...
```

Figure A.6 analysis template for CI

As Shown in Figure A.6, the 'Causes' field indicates the possible values or limits for each input and the 'Effects' field indicates all the possible range of the return value. The 'Truth Table' gives a good view to show all the possible combinations of inputs to create related return value. In the truth table, each rule is a test case. Based on these CI test cases, combined with the tested method's inputs, the CI test data of this method can be produced, just as shown in Figure A.7.

```
//  Test Data:
//         Test Cases              Inputs            Expected Outputs
//    ID    Covered          xxx   xxx   xxx  ...      return value
//    Test1 CI1              ...   ...   ...  ...      ...
//    Test2 CI2              ...   ...   ...  ...      ...
//    Test3 CI3              ...   ...   ...  ...      ...
//    ...   ...              ...   ...   ...  ...      ...
```

Figure A.7 CI test data generation

2. Test Realization

Based on the test data for different testing techniques generated in step 1, the detailed code for each test can be realized in this stage. The code template of black-box test method is shown in Figure A.8.

```
@MyTest
public static boolean testMethod_Technique_Txx() {
    //call the tested method and obtain the actual return value
    if(actual_return_value == expected_return_value)
        return true;
    else{
        reason = new String("Expected 'expected_return_value', actual result :"+
actual_return_value);
```

```
        return false;
    }
}
```

Figure A.8 test method template

The annotation '@MyTest' indicates that this method is a test method and can be found and executed by customized JUnit test runner to collect test result. The name of test method is varied based on the test technique used (EP -> testMethod_EP_Txx, BVA -> testMethod_BVA_Txx, CI-> testMethod_CI_Txx). For the inside code, just call the tested method based on the related test data and compare the actual return value with the expected return value. If they are the same, just return true to indicate this test is passed, otherwise show the reason why it is wrong and return false.

3. Test Execution

The tests were executed using a customized test runner and the results collected to prepare for the final analysis and comparison.

☐ Black-Box test implementation example

1. Three Techniques Test Cases Analysis and Tests Generation

☐ EP for `Login()`

Based on the EP analysis and tests generation template together with the specification of `Login()`, the detailed EP analysis and tests generation for `Login()` method is shown in Figure A.9. Each 'Email' and 'Password' partition is a test case. Based on these 5 test cases, 3 EP tests are generated.

```
//[Test Cases] Partition {Test}
// Inputs
//    Email: [EP1] correct format but not existing email string {T001}
//           [EP2] correct format and existing email string     {T002}
//    Password: [EP3] wrong format password string              {T003}
//    [EP4] correct format but not 'correct' password string    {T001}
//    [EP5] correct format and 'correct' password string        {T002}
//
// Outputs
//    return values:
//    [EP6] "{'result': 0}"                                    {T001}
//    [EP7] "{'result': 1, 'id': .., 'username': ..}"          {T002}
//
// [Test Data]
//     Test Cases              Inputs                Expected Outputs
// ID   Covered       Email            Password      return value
```

```
//T001 EP1,EP4,EP6 linanpp01@gmail.com  12345678      "{'result': 0}"
//T002 EP2,EP5,EP7 linanpp09@gmail.com  123456
//                    "{'result': 1, 'id': 1, 'username': 'Li Nan'}"
//T003 EP3*         linanpp10@gmail.com 1234,;--=     "{'result': 0}"
//
// Note: * indicates an error test
```

Figure A.9 EP analysis and test generation for Login()

- BVA for `Login()`

  Based on the BVA analysis and tests generation template together with the specification of `Login()`, the detailed BVA analysis and tests generation for `Login()` method is shown in Figure A.10. The 'Email' and 'Password' boundary value are the same to their partition value, so BVA and EP have the same tests for testing `Login()` method.

```
//[Test Cases] Boundary Value {Test}
// Inputs
//    Email: [BVA1] correct format but not existing email string {T001}
//           [BVA2] correct format and existing email string     {T002}
//    Password: [BVA3] wrong format password string              {T003}
//    [BVA4] correct format but not 'correct' password string    {T001}
//    [BVA5] correct format and 'correct' password string        {T002}
//
// Outputs
//    return values:
//    [BVA6] "{'result': 0}"                                    {T001}
//    [BVA7] "{'result': 1, 'id': .., 'username': ..}"          {T002}
//
// [Test Data]
//     Test Cases                Inputs              Expected Outputs
// ID  Covered       Email                Password   return value
//T001 BVA1,BVA4,BVA6 linanpp01@gmail.com  12345678    "{'result': 0}"
//T002 BVA2,BVA5,BVA7 linanpp09@gmail.com  123456
//                      "{'result': 1, 'id': 1, 'username': 'Li Nan'}"
//T003 BVA3*          linanpp10@gmail.com  1234,;--=   "{'result': 0}"
//
// Note: * indicates an error test
```

Figure A.10 BVA analysis and test generation for Login()

- CI for `Login()`

  Based on the CI analysis and tests generation template together with the specification of `Login()`, the detailed CI analysis and tests generation for `Login()` method is shown in Figure A.11. Besides the 3 same tests to EP and BVA, CI finds out another test to check whether `Login()` meets its

specification or not.

```
// Causes:
// 1. Email -> correct format but not existing email string
// 2. Email -> correct format and existing email string
// 3. Password -> wrong format password string
// 4. Password -> correct format but not 'correct' password string
// 5. Password -> correct format and 'correct' password string
//
// Effects:
// 1. "{'result': 0}"
// 2. "{'result': 1, 'id': .., 'username': ..}"
//
// [Test Cases] Truth Table {Test}
//                                                  Rules
//                                        [CI1]   [CI2]   [CI3]   [CI4]
//  Causes:
//  Causes -> 1                             T       F       F       *
//  Causes -> 2                             F       T       T       *
//  Causes -> 3                             *       F       F       T
//  Causes -> 4                             *       T       F       F
//  Causes -> 5                             *       F       T       F
//  Effects:
//  Effects -> 1                            T       T       F       T
//  Effects -> 2                            F       F       T       F
//                                        T001    T004    T002    T003
//                                                  Tests
//
// [Test Data]
//       Test Cases          Inputs                  Expected Outputs
//  ID    Covered    Email            Password        return value
//  T001  CI1   linanpp01@gmail.com    12345678        "{'result': 0}"
//  T002  CI3   linanpp09@gmail.com    123456
//                     "{'result': 1, 'id': 1, 'username': 'Li Nan'}"
//  T003  CI4   linanpp10@gmail.com    1234,;--=       "{'result': 0}"
//  T004  CI2   linanpp09@gmail.com    1234567         "{'result': 0}"
```

Figure A.11 CI analysis and test generation for Login()

2. Tests Realization

Based on the three techniques analysis on `Login()` in step 1, 4 tests are generated: T001, T002, T003 and T004. The detailed codes for these four tests are shown in Figure A.12.

```
@MyTest
public static boolean testLogin_EP_T001() {
   String result = Provider_Interface.Login("linanpp01@gmail.com", "12345678");
   if(result.equals("{'result': 0}"))
         return true;
   else{
         reason=new String("Expected {'result': 0} , actual result :"+result);
         return false;}
}

@MyTest
public static boolean testLogin_EP_T002() {
    String result = Provider_Interface.Login("linanpp09@gmail.com", "123456");
    if(result.equals("{'result': 1, 'id': 1, 'username': 'linanpp34'}"))
         return true;
    else{
         reason=new String("Expected {'result': 1, 'id': 1, 'username':
'linanpp34'} , actual result :"+result);
         return false;}
}

@MyTest
public static boolean testLogin_EP_T003() {
  String result = Provider_Interface.Login("linanpp10@gmail.com", "1234,;--=");
  if(result.equals("{'result': 0}"))
         return true;
  else{
         reason=new String("Expected {'result': 0}, actual result :"+result);
         return false;}
}

@MyTest
public static boolean testLogin_CI_T004() {
    String result = Provider_Interface.Login("linanpp09@gmail.com", "1234567");
    if(result.equals("{'result': 0}"))
            return true;
    else{
            reason=new String("Expected {'result': 0}, actual result :"+result);
            return false;}
}
```

Figure A.12 4 Black-Box tests on Login()

Each test calls `Login()` by using the analyzed test data, if the return result does not equal to the expected value, then the test will show the reason and return false to

show the `Login()` does not have correct reaction on this test. By using such code, the detailed reason about why `Login()` fails can be viewed in the console window.

3. Test Execution

Finally, just use my test runner to run these 4 Black-Box tests, the calling code is shown in figure A.13.

```
public static void main(String[] args)
{
        MyTestRunner r = new MyTestRunner();
        r.run("Internship_Test.Provider_InterfaceTest");
}
```

Figure A.13 run Black-Box tests to test Login()

By using this code, the 4 tests will be executed to check whether `Login()` method meets its specification or not. Console output gives the execution result and Eclemma gives the Black-Box testing coverage result of this method, just as shown in Figure A.14, A.15 and A.16.

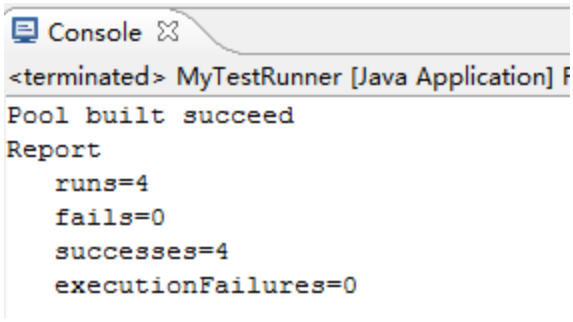


Figure A.14 Black-Box testing result on Login()

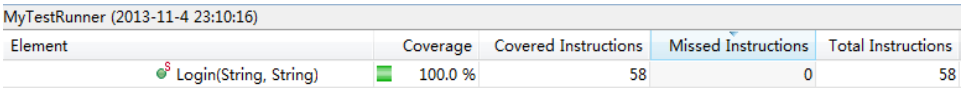


Figure A.15 Black-Box testing code coverage on Login()

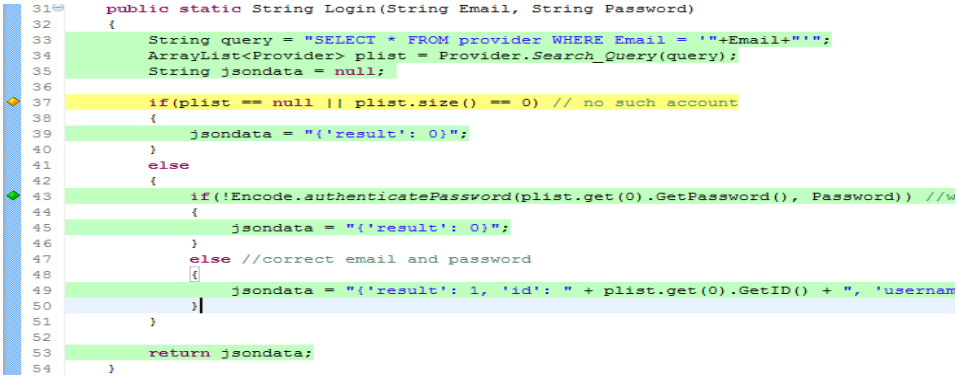


Figure A.16 Black-Box testing branch coverage on Login()

Figure A.14 indicates that all 4 Black-Box tests have passed, which means the method `Login()` actually meets its specification requirement. Figure A.15 and A.16 indicate its coverage result: 100% code coverage and 75% branch coverage, which means that although the line 37 was executed, the statement 'plist == null' was never equal to true. These two points together with other data (for example: the number of Black-Box tests is 4) can be collected to prepare for the final analysis on `Login()` method.

Appendix B Calculate the number of complex Boolean expressions with n conditions

When start to do the MC/DC testing on other Java programs, the first step aims to calculate the numbers of complex Boolean expressions with n (1, 2, 3…) conditions inside the tested program. Because of the large amount of source code, it is difficult to calculate these numbers manually. A better way is to use Java byte code.

For example, the source code of method `testjavabytecode()` is shown in Figure B.1 and its byte code is shown in Figure B.2.

```
public int testjavabytecode(int i, int d, int c)
{
    if((i< 20 || d > 30) && c > 10)
    return 1;
    else
    return 0;
}
```

Figure B.1 source code of method testjavabytecode()

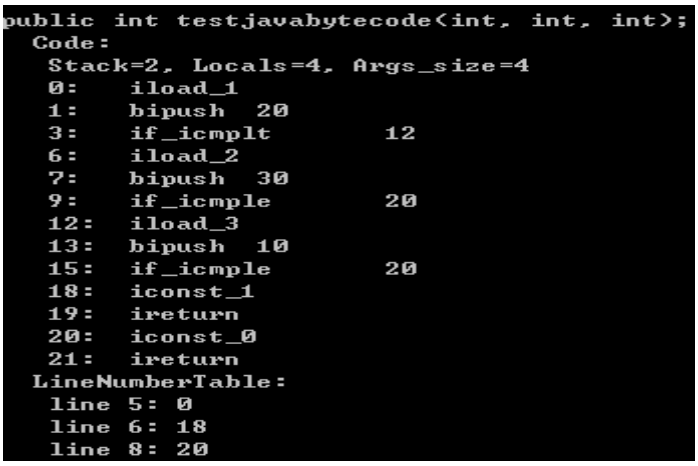


Figure B.2 byte code of method testjavabytecode()

As shown in Figure B.2, the '`Code`' part is the generated instructions list and the '`LineNumberTable`' part indicates the line-instruction information of this method. These two parts information can be used to automatically calculate the number of complex Boolean expressions with n decisions. The calculating algorithm is:

1. Read '`Code`' part line by line and store each instruction into a string 'instruction' array (for this method, create a string array with 21 elements, store instruction 0 into array position 0, instruction 1 into array position 1, instruction 3 into array position 3 and so on)

2. Read '`LineNumberTable`' part line by line and store line-instruction information into an integer 'line-instruction' array (for this method, the 'line-instruction' array is: [0]->0, [1]->18, [2]->20).

3. Search and count the number of jump instructions in 'instruction' array between the adjacent value of 'line-instruction' array (for example, because 'line-instruction' array[0] = 0 and 'line-instruction' array[1] = 18, so search jump instructions in 'instruction' array between 'instruction' array[0] to 'instruction' array[18]).

4. By performing the step 3, jump instructions 'instruction' array[3] (`if_icmplt`), 'instruction' array[9] (`if_icmple`) and 'instruction' array[15] (`if_icmple`) are found, which means the method `testjavabytecode()` has a complex Boolean expressions with 3 conditions.

By using such algorithm, the number of complex Boolean expressions with n decisions in the tested programs could be calculated automatically. For the detailed testing part, the testing step is the same to that of testing COMWS and the result can be collected to evaluate whether MC/DC is still effective in testing those non state-based programs.